VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Implementace složitějších kódů proměnné délky

Implementation of Advanced Variable-length Codes

2011 **Pavel Sidžina**

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Pavel Sidžina**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace složitějších kódů proměnné délky
Implementation of Advanced Variable-length Codes

Zásady pro vypracování:

Cílem práce je nastudovat a implementovat několik pokročilejších typů kódování pro symboly/čísla, porovnat je z hlediska efektivity a hlavně připravit je pro použití v kompresním frameworku.

Postupujte dle následujících bodů:
1. Nastudujte následující typy kódování:
   a) Boldi-Vigna kódy.
   b) Yamamotovy rekurzivní kódy.
   c) Tabu kódy.
   d) Wangovy a Yamamotovy Flag kódy.
2. Implementujte jednotlivá kódování.
3. Otestujte jejich efektivity pro různé druhy dat.
4. Sestavte moduly pro jednotlivá kódování pro použití v kompresním frameworku.

Seznam doporučené odborné literatury:

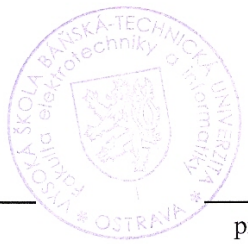
Variable-length codes for Data Compression, David Salomon, Springer, 2007
Data Compression: The Complete Reference, David Salomon, 4. ed., Springer, 2007

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Platoš, Ph.D.**

Datum zadání: 20.11.2009

Datum odevzdání: 06.05.2011



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou/diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Souhlasím se zveřejněním této bakalářské/diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava.

V Ostravě dne 3.5.2011 …….

Zde bych chtěl poděkovat vedoucímu této práce, Janu Platošovi za trpělivost a pomoc při řešení problemů.

ABSTRAKT

Cílem této práce je implementace kompresních kódů, otestování a celkové zhodnocení jejich efektivity. Kódy, kterými se zabývám, jsou Boldi-Vigna kódy, Tabu kódy, Yamamotovy rekurzivními kódy, Yamamotovy flag a také Wang flag kódy. Testovány jsou pro kladná čísla v rozsahu 2^0 až 2^{31} a následné výsledky porovnávány s klasickým 32bitovým zápisem.

Pro porovnání efektivity kompresí kódů je nejdříve potřeba nelézt nejúčinnější varianty kódů. Důležité informace nám dodá také prozkoumání efektivity nejen v celém spektru, ale i v speciálních rozsazích.

KLÍČOVÁ SLOVA

Boldi-Vigna kódy , Yamamotovy rekursivní kódy , Tabu kódy, Wangovy a Yamamotovy flag kódy, implementace kódů , testování efektivity

ABSTRACT

The aim of this work is implementation and testing of compression codes, and exaluating their effectivity. Codes, on which i work and test, are Boldi-Vigna codes, Taboo codes Yamamoto recursive codes, Yamamoto flag codes and Wang flag codes. They are tested for positive numbers in range 2^0 to 2^{31} and results of codes are compared with classic 32bits record of number.

First thing in comparing effectivity of compression codes is searching for the most effective variant of code. Exploring effectivity in our entire spectrum and also in exact ranges gives us significant informations too.

KEY WORDS

Boldi-Vigna codes, Yamamoto recursive codes, Taboo codes, Wang and Yamamoto flag codes, implementation of codes, testing of efectivity

1. Úvod

Složité kompresní kódy proměnné délky mají v našem světě plného přenosu a uchovávání velkého množství dat nemalý význam a děje tak i přesto, že výrobci pevných disků přichází na trh s terabajtovými kapacitami a poskytovatelé internetu se předhání v rychlostech downloadu i uploadu. I když před lety byly kapacity a přenosové rychlosti ohromně menší, a data bylo třeba co nejvíce komprimovat, dnes se zase soubory pyšní ohromnou velikostí, což je důsledek většinou větších kvalit filmů a fotek, či přepracováváním velké části psaných či jiných informací do elektronické podoby včetně databází, obsahujících nepřeberné množství dat na internetu. A právě zde opět nachází své uplatnění kompresní kódy, které pomocí speciálně vymyšlených vzorců či logiky dokážou robustné data odlehčit a usnadnit tak jejich uskladnění.

Ve své bakalářské práci se zabývám kódy podporující myšlenku bezeztrátové komprese, které na rozdíl od ztrátové komprese, nepřijdou nenávratně o své data a uchová je v 100% kvalitě. Nevýhoda tohoto přetvoření je časová náročnost jak při kompresi, tak při dekompresi a také fakt, že pro použití dat je třeba data dekomprimovat. Časová náročnost však neovlivňuje efektivitu komprese, jak by se na prvý pohled mohlo zdát. Hlavní roli mají již dříve zmíněné vzorce a také šikovnost programátora. Dalším velmi důležitým faktorem jsou sama data. Každá logika použita v kompresních kódech má své účinnější a méně účinné části, které závisí jak na výběru oddělujícího kódu, tak na velikosti čísla vyjádřeného v binárním kódu. Většinou se setkáváme s tím, že kódy, kterým se velice efektně daří zakódovat malé čísla, mívají velmi malý účinek na čísla vysoká apod. Zde se budu zabývat určitými kódy a to Boldi-Vigna kódy, Tabu kódy, Wrang a Yamamotovy flag kódy a také Yamamotovými rekurzivními kódy.

V praxi se při těchto vlastnostech nejlépe uplatňují celkově nejrychlejší a nejefektivnější komprese a kompromisní varianty, kdy například existuje kód, který komprimuje efektivně, ale za cenu extrémně vysoké časové náročnosti. V těchto případech se mohou spíše uchytit kódy, které komprimují méně, avšak stále velice efektivně, ale zato za menší čas. Na speciálních místech, kde se předem ví, jaká data budou na vstupu, a jedná se o data pouze z určité škatulky, se použijí kódy nejvíce vyhovující daným datům.

2. Kódy

V této kapitole se věnuji všem dříve zmíněným kódům, jejich konstrukci a následným dekódováním čistě v teoretickém duchu.

2.1. Boldi-vigna kódy

Boldi-Vigna kódy jsou vytvořeny pány Paolo Boldi a Sebastiano Vigna. Jejich zeta kód je součástí ukládání a komprese webových grafů využívaných internetovými vyhledávači.

Ve zkratce, ve světě posedlým internetem mají velkou důležitost vyhledávače. Webový graf, který využívají, si můžeme představit jako graf s uzly propojený hranami, kde uzly jsou stránky samotné a hrany spojující odkazy. Když vezmeme v potaz, že těchto uzlů je stovky miliónů a odkazů několik biliónů, navíc jejich počet se exponenciálně zvětšuje, je třeba, aby takové množství dat bylo efektivně uloženo.

Jakožto efektivní řešení se zvolilo takové, že každá URL je reprezentovaná a uložená jako celé číslo v seznamu adres a odkazy na které ukazuje, se nachází v dalším seznamu, kde se odkazy, pro lepší kompresi, ukládají jako rozdíl URL rodiče a potomka. Poté přichází další praktiky pro nejlepší možnosti komprese, až přichází na samotnou kompresi Boldi-Vigna zeta kódem, který se vyhodnotil jako nejlepší řešení pro kompresi webových grafů.

2.1.1. Logika tvorby kódu

Boldi-Vigna je kód, založen spíše na vzorcích než na logice generování kódu s oddělovači. Možná právě toto je jeho klíč ke slávě, v podobě jednoduchosti tvorby kódu a díky propočtům pomocí vzorců přináší rychlejší zpracování. Jeho konstrukce je poměrně jednoduchá, založená na několika vzorcích. Máme zde kladné celé číslo *k*, což bude pro nás kompresní faktor. Díky tomuto faktoru si vytvoříme sety čísel v rozmezích $\left[2^{hk}, 2^{(h+1)k} - 1\right]$, kde h je kladné číslo indexující tyto sety čísel od 0. Naše číslo, které budeme komprimovat, budeme nazývat *n*.

Nyní, abychom zjistili hodnotu *h*, kterou budeme v dalších krocích potřebovat, je třeba zjistit, do kterého setu čísel náleží *n*. K tomuto postačí nalézt rozsah setu, do kterého *n* patří a to pod podmínkou $2^{hk} \leq n \leq 2^{(h+1)k} - 1$. Pokud *n* splňuje tuto podmínku, budeme pro naše číslo v budoucnu používat tuto hodnotu *h*. Nyní si vypočteme minimální binární kód *x*, který nás dovede k výpočtu, kolik bitů bude použito v kódu. Rovnice na výpočet *x* je $x = n - 2^{hk}$. Další, co budeme potřebovat, je délka intervalu *z*, což je $z = 2^{(h+1)k} - 2^{hk}$. Teď si musíme spočíst, kolik bitů budeme potřebovat na kód. Na to přijdeme vypočtením rovnice $s = \log_2 z$.

Jak jsme uvedli pár řádku zpět, hodnota *x* má možnost ovlivnit počet bitů, ale je také součásti výsledku. K dalšímu postupu slouží tato nerovnice: $x < 2^s - z$. Pokud je pravdivá, prozatímní výsledek *V* bude hodnota *x* na *s-1* bitech. V opačném případě je výsledek $x - z - 2^s$ na *s* bitech. Ne nadarmo jsem však uvedl, že se jedná o prozatímní výsledek, protože se před tento kód ještě vloží unární zápis *h*, což je $0^h 1$, neboli *h*-krát nula zakončena jedničkou. Nyní jsme dosáhli kýženého výsledku, avšak postup může působit chaoticky, a z vlastní zkušenosti vím, že nejlepší je projít si konkrétní příklad.

2.1.2. Dekódování

Dekódování se táhne ve stejném duchu jako zakódování, prakticky jde o poměrně opačný postup. Na vstupu budeme vyčkávat, dokud nenarazíme na *1* a při tom počítat kolik *0* předcházelo této jedničce. Tento počet bude naše *h*. Nyní, jelikož *k* je známo, můžeme vypočíst zbytek hodnot.

Nejdříve si vytvoříme hranice našeho rozsahu setu čísel. K tomu nám pomůže opět tento vzorec $[2^{hk}, 2^{(h+1)k} - 1]$. Opět vypočteme délku intervalu *z*, jako rozdíl horní a spodní hranice rozsahu. Díky spočítání *z* teď máme možnost spočítat, na kolika bitech by měl být uložen náš kód, což zjistíme ze vzorce $s = \log_2 z$. Avšak zde přichází problém s počtem bitů. Při zakódování se ukládá číslo buď na *s* bitů nebo na *s-1* bitů. Proto si nejdříve načteme s-1 bitů a vyzkoušíme, zda nějaké číslo z našeho rozmezí dokáže být zakódováno do takovéhoto kódu. Pokud nenalezneme žádné takové číslo, načteme jeden bit navíc, a zde si můžeme být jistí, že hledané číslo nalezneme.

2.2. Yamamotovy rekurzivní kódy

Psal se rok 2000, ve kterém přišel Hirosuke Yamamoto s geniální a jednoduchou inovací již používaného Eliasova omega kódu. Tento kód pracuje na principu vkládání délkových prefixů, čímž si vysloužil přezdívku „rekurzivní Eliasův kód“, podle čehož se nazval také Yamamotova verze kódu.

Eliasův omega kód pracuje na logice, kdy vezme číslo a před něj vloží délku tohoto čísla. Poté vezme délku této délky, vypočte její délku a opět ji vloží dopředu, dokud je to možné. Vznikne tedy kód rozdělený do několika skupin, které vždy začínají jedničkou. Aby se rozeznal konec kódu, vloží se za celý kód nula. Yamamoto si však všiml, že tento styl kódování plýtvá bity, a to již dříve zmíněnými jedničkami, jelikož se dobře ví, že zde budou vždy.

Přišel tedy s řešením, které však stěžuje zakódování čísla a rozšiřuje velikost oddělovače, ale zato příliš neplýtvá bity.

2.2.1. Logika při tvorbě kódu

Yamamotův rekursivní kód sice vychází z Eliasového omega kódu, ale podoba při tvorbě kódu zůstává pouze v hlavní myšlence rekurzivity s délkami. Když to vezmeme odzadu, výsledný kód bude vypadat symbolicky například takto – *abbcccc(DEL),* kde *DEL* je oddělovač (anglicky delimiter) pro rozeznání konce kódu a následné vyhledání zakódovaného čísla, které oddělovači předchází, tedy seskupení znaků *c*. Dále zde máme seskupení znaků *b*, které je binární vyjádření bitové délky znaků *c*, a poté je tu znak *a*, který reprezentuje bitovou délku znaků *b* - tedy délku délky. Tato ukázka je pouze informativní, pro naznačení struktury kódu, kód samozřejmě může obsahovat méně či více délkových skupin o různých délkách ale o tom níže.

Jako první věc, kterou začneme vytvářet kód, je oddělovač. Jeho délku budeme označovat znakem *f* a jeho hodnotu budeme udávat jako *del*. Nejpoužívanější délky oddělovače jsou 2 a 3 bity. Co za kombinaci čísel si zvolíme v oddělovači je na nás, v podstatě to má minimální význam.

Poté si vezmeme to naše kýžené číslo, které chceme zakódovat a budeme jej označovat písmenkem *n* a jeho délku v bitech jako n_d. Abychom zvětšili naše možnosti vyjádření čísel pro kódování, budeme potřebovat neklasické vyjádření. Čísla nebudou reprezentovány správně binárně 0,1,01,10,11,100 atd., ale budou formulovány pomocí kombinací čísel 0 a 1, tedy 0,1,00,01,10,11,000 atd. Abychom však docílili toho, že bude možné pomocí oddělovače rozeznat

konec kódu, potřebujeme některé kombinace vyškrtat, jelikož by zapříčinily nemalé problémy s dekódováním.

Když se na budoucí kód, který bude vypadat např. takto - *abbcccc(DEL)*, podíváme, můžeme přijít na to, proč se musí některé varianty vypustit. Jak jsem již nastínil, jde o konflikt s nalezením oddělovače při dekódování. Dekodér bude dopředu vědět jaký oddělovač hledat a kdyby na něj narazil moc brzo, došlo by k ukončení vyhledávání. Za hledané číslo by se zvolila předchozí skupina čísel. Z tohoto důvodu potřebujeme vypustit kombinace, které na svém začátku mají kombinaci oddělovače. To však platí pouze pro kombinace, jejichž délka v bitech je $\geq n_d$. Pro délky $< n_d$ platí, že se vypustí ty, které jsou součástí začátku oddělovače, např. 01 u oddělovače 011 atp. ne však 11.

Po tomto definování špatných kombinací můžeme zbylé čísla prohlásit naší číselnou řadou. Tímto se dostáváme do další fáze kódování. Máme naše číslo *n* a nalezneme k němu správnou binární kombinaci, kterou nazveme K_n .To nám však k zakódování čísla nestačí, protože v tomto kódu se může jinde než na začátku ukrývat hodnota oddělovače. Proto si vypočteme bitovou velikost K_n , odečteme 1, a této hodnotě najdeme opět kód v naší číselné řadě. Jakmile tak učiníme, předsadíme tento kód před K_n . Zde ale s kódem opět narážíme na stejný problém, avšak se stejným řešením. Vezmeme si kód, který jsme předsadili a spočítáme jeho bitovou délku, odečteme 1, a nalezený kód zase předsadíme před vše. Takhle pokračujeme, dokud se nedostaneme k bitové délce o hodnotě 1.

Nyní jsme se dostali k výslednému kódu, který může být ve formě *abbcccc(DEL)*, kdy už víme, že *a* je zakódovaná délka znaků *b*, které jsou zakódovaná délka znaků *c*, což je zakódované naše číslo následované oddělovačem pro snadné dekódování.

2.2.2. Dekódování kódu

Dekódování našeho kódu se dá lehce logicky odvodit z předchozího postupu. Začneme od začátku. Jakožto dekodér víme, jak má vypadat oddělovač a proto se jej budeme snažit při každém postupu hledat. Nicméně na vstupu se nám ukáže vždy první hodnota, která oddělovač být nemůže. Je to bit *a*, který nám udává zakódovaně buď velikost další skupiny znaků + 1, či už to je naše hledané číslo. To poznáme díky zkoumání, zda se za tím nachází už oddělovač či jiné znaky. Pokud se zde nenachází oddělovač, díky odkódovaní *a* víme, jak dlouhá bude další skupina znaků.

Takto to pokračuje dál, dokud nenarazíme na oddělovač. Jakmile jej nalezneme, víme, že poslední dekódované číslo bylo naše hledané číslo.

2.3. Tabu kódy

Tabu kódy jsou myšlenkou pana Stevena Pigeona. Jedná se o poměrně jednoduché kódy tvořené výhradně vzorcem. V mé bakalářské práci se po domluvě s panem Platošem budu zabývat pouze block-based verzí kódu.

Kód je poměrně jednoduchý, tvořený stejně dlouhými bloky bitů o stejné délce, a blokem na konci s unikátním setem bitů, jež zde slouží jako oddělovač. Tento kód, na rozdíl od jiných kódů, počítá s kladnými čísly a i s nulou. Co se týče tvorby tohoto kódu, dal by se tvořit jednoduše inkrementacemi čísel reprezentující bloky, avšak trvalo by velmi dlouho, než by se takto vypočítala vysoká čísla. Proto byl vytvořen vzorec na vztah mezi různými bloky a čísly, které dohromady reprezentují výsledek, jež jednoznačně pracuje rychleji než první nastíněný postup.

2.3.1. Logika při tvorbě kódu

Jak již bylo řečeno, kód je tvořen stejně dlouhými bloky bitů zakončený oddělovacím blokem. Velikost těchto bloků určíme při zakódování a budeme ji nazývat *m*. Velikost není omezena, jelikož můžeme využít $m > 0$, avšak logicky budeme určovat $m > 1$, jelikož bychom při velikosti 1 tvořili nesmyslně dlouhé řetězce stejných bitů.

Dále si určíme oddělovač, tedy jeho hodnotu o velikosti *m*. Hodnota je však bezvýznamná a pro lehčí konstrukci kódu a pochopení budeme používat oddělovač o hodnotě 0^m. Díky tomuto víme, že v blocích čísla budeme tvořit hodnoty pomocí kombinací nul a jedniček ovšem bez první hodnoty kombinací, tedy bez samých nul. Tedy např. pro $m=2$ bude oddělovač 00 a bloky budou moct tvořit hodnot 01, 10 a 11. Jeden blok nám tedy dokáže vyjádřit 3 hodnoty, tudíž pro čísla 0-2. Pro číslo 4 budeme muset vytvořit další blok bitů, který bude spolu s předešlým blokem sloužit pro čísla 3-11, kdy se pomocí těchto bloků tvoří všechny možné kombinace. Zkrátka každý blok o velikosti *m* dokáže vytvořit 2^m-1 použitelných kombinací. Pokud budeme mít bloků za sebou *k*, tak počet kombinací o *k* blocích je $(2^m-1)^k$.

Jak jsem již ale naznačil, nejrychlejší a nejelegantnější postup je řešení pomocí vzorců, a proto navážeme na náš předešlý vzorec, k výpočtu jej totiž budeme potřebovat. Abychom věděli, kolik bloků na naše číslo *n* budeme potřebovat, musíme si vypočítat speciální vzorec. Jelikož již známe vzorec na počet čísel, která jdou znázornit určitým počtem bloků, stačí nám postupně sčítat tyto hodnoty pro 1..2..3..*k* bloků, dokud součet nebude větší, než naše číslo *n*. Tento součet budeme nazývat součet geometrické progrese a budeme jej značit jako $g_m(k)$, kde *k* je počet těchto bloků. $g_m(k)$ lze vyjádřit tímto vztahem:

$$g_m(k) = \frac{[(2^m - 1)^k - 1](2^m - 1)}{(2^m - 2)}$$

Při výpočtu tedy budeme inkrementovat *k* a čekat dokud výsledek nebude vyhovovat této nerovnici:

$$n \leq g_m(k) - 1$$

Jakmile splníme vztah, víme, že náš kód bude tvořen $(k+1)n$ bity v *n* blocích včetně oddělovače. Nyní potřebujeme pro další výpočty vypočítat proměnnou, kterou si nazveme *c*, ze které vypočteme hodnoty pro každý blok bitů. Vypočteme ji ze vztahu:

$$c = n - g_m(k-1)$$

Teď už k samotné tvorbě bloků. Pomocí cyklu budeme vypočítávat hodnotu každého bloku a k této hodnotě nalezneme kód, ke kterému patří. Hodnotu bloku budeme nazývat b_i, kde *i* je pořadí bloku od oddělovače tudíž: $b_2b_1b_0DEL$.Pro výpočet b_i použijeme tento vzorec:

$$bi = \left[\left\lfloor \frac{c}{(2^m - 1)^i} \right\rfloor \% (2^m - 1)\right] + 1 \qquad kde\ i = 0,1,..k-1$$

Jakmile známe hodnoty ke každému bloku, stačí nám k nim nalézt hodnoty a ty naskládat ve správném pořadí za sebe. Tyto hodnoty, jak bylo uvedeno výše, jsou tvořeny kombinacemi nul a jedniček. Jejich pořadí si můžeme představit tak, že minimální hodnota jsou samé nuly a

maximální samé jedničky, s tím že k nulám připočítáváme binárně jedničky. Nakonec z těchto hodnot musíme ještě vypustit hodnotu kódu oddělovače, protože tu přidáme pouze za celý kód, a tudíž se nesmí zobrazit jinde než na konci kódu.

2.3.2. Dekódování kódu

Dekódování máme ještě jednodušší než zakódování. Před tím než se dostaneme do styku s kódem, je nám známa velikost bloků *m* a také hodnota oddělovače. Bity načítáme po blocích a zkoumáme, zda kód bloku nesouhlasí s kódem oddělovače. Jakmile narazíme na tento kód, budeme pracovat již jen s dosud načtenými hodnotami.

Kódy bloků se hodí převést na hodnoty *b*, kterým odpovídají. Hodnoty si můžeme odložit do dočasného pole v takovém pořadí, abychom nezaměnili směr a věděli, které pozici patří index *i*= 0. Po tomto seřazení si v cyklu spočteme již výsledné číslo podle vzorce:

$$n = g_m(k-1) + \sum_{i}^{k-1} (b_i - 1)\,(2^m - 1)^i$$

2.4. Wangovy flag kódy

Podobně jako tomu je u Taboo kódů, tak i Wangovy flag kódy jsou založeny na oddělovači tvořeným nulami. Struktura tvorby kódu velmi jednoduše zaručuje nemožnost nalézt oddělovač vně kódu a zároveň svou jednoduchou konstrukcí podporuje časově rychlé tvoření kódů.

2.4.1. Logika při tvorbě kódu

Tvorba kódu je opravdu jednoduchá. Na začátku si určíme velikost oddělovače $f \geq 2$, který nám bude říkat, z kolika nul se bude oddělovač skládat. Dále zde budeme mít naše číslo *n*, které chceme zakódovat.

První krok, je že naše číslo *n* převedeme do binární podoby a následně jednotlivým bitům převrátíme pořadí. Jednodušeji by se to dalo říct, že bity zapíšeme zprava doleva. Docílíme tak obráceného pořadí bitů. Následně projdeme sekvenci bitů a budeme zde hledat *f-1* nul za sebou. Až narazíme na tento případ, vložíme za ně jedničku, čímž se zaručí, že se uvnitř kódu oddělovač nenajde. Jakmile se dostaneme na konec, vložíme za tyto bity set nul oddělovače a zapečetíme tak finální podobu kódu.

2.4.2. Dekódování kódu

Dekódování kódu je také velmi jednoduché a lze vyvodit z postupu při zakódování. Při načítání bitů pokračujeme, dokud nenarazíme na bity oddělovače. Vezmeme si tedy set, který jsme načetli, odstraníme z něj oddělovač a zbytek bitů zpracujeme. Nejdříve projdeme bity zleva doprava a budeme vyhledávat *f-1* nul za sebou. Jakmile je nalezneme, odstraníme jedničku za nimi, o které 100% víme, že za nimi následuje. Takto projdeme celý zbytek setu bitů až do konce. Potom stačí jen bity zapsat v opačném pořadí a tímto získáme binární hodnotu našeho hledaného čísla *n*.

2.5. Yamamotovy flag kódy

Yamamotovy Flag kódy jsou jistým způsobem vylepšení pro Wang Flag kódy. Ty u své tvorby potřebují reverzi bitů, a jelikož první bit čísla je vždy 1, nalezneme jej vždy před oddělovačem, který je vždy tvořen nulami, čímž nám zaručí lehké nalezení oddělovače. Na rozdíl od Wangova kódu, je kód Yamamota a Ochiho (autorů) poněkud složitější. Avšak v celkovém výsledku má kladný dopad na rychlost jak při zakódování, tak i pro dekódování. Navíc si uvědomili, že už dříve

zmíněná jednička je u čísel vždy, a je tedy logické ji vypustit a přenést váhu pro nalezení oddělovače jinde.

2.5.1. Logika při tvorbě kódu

Jak již bylo řečeno, logika kódu je trochu složitější než u Wangova kódu, ale pochopit se dá lehce. Začínáme opět s naším číslem *n*, které chceme zakódovat, a s určením oddělovačem *f*. Jeho délka by měla být minimálně 3. Jeho hodnota musí však být $0^1 1^{f-1}$ a $1^1 0^{f-1}$, nebo $0^2 1^{f-2}$, $0^2 1^{f-2}$ pro $f > 3$, aby se zaručilo, že se oddělovač nevytvoří, tam kde nemá.

Bity, které tvoří oddělovač, budeme označovat indexy, tedy $f = \{f_1, f_2, f_3 \dots f_i\}$, a bity, co tvoří naše číslo $n = \{n_1, n_2, n_3 \dots n_i\}$. U čísla však bereme pouze velikost použitých bitů, např. pro 5 jsou bity 3 apod. Jak již bylo naznačeno dříve, první bit n_1 je vždy 1, což je dopředu známo, tím pádem jej vypustíme.

Nyní máme vše přichystáno pro tvorbu kódu. Dále projdeme bity čísla *n* a budeme hledat, kde se nachází kombinace bitů na f_1 až f_{i-1} kódu oddělovače. Pokud tuto kombinaci nalezneme, vložíme za ní negovanou hodnotu oddělovače f_i. Tímto vložením negace za kód zaručíme, že kód půjde dekódovat zpět a vyloučíme hodnoty oddělovače na nevhodných místech. Jakmile takto projdeme celé číslo, připojíme za něj hodnotu oddělovače, čímž to pro nás končí.

2.5.2. Dekódování Kódu

Jakmile pochopíme logiku tvorby kódu, dekódování nám může být jasné. Při načítání bitů vyčkáváme, dokud nenalezneme kód oddělovače na správném místě. Jakmile jej najdeme, vyloučíme ho ze setu bitů.

Musíme si však uvědomit, že vložením bitu při zakódování můžeme nechtěně vytvořit kombinaci oddělovače s následujícími bity, proto musíme být u zpracování hodně pozorní.

Pro naše číslo již víme, že první bit $n_1 = 1$. Poté budeme procházet náš set bitů a prohledávat bity za sebou o velikosti *i-1*, zdali náhodou nenalezneme shodu s oddělovačem bez posledního bitu. Pokud shodu nalezneme, vyškrtneme následující bit ze setu. Po tomto průchodu a odstranění nadbytečných bitů je výsledný set čísel binární zápis našeho hledaného čísla *n*.

3. Implementace kódů

Pro lepší pochopení postupů tvoření kódů jsou nejlepším způsobem názorné příklady, které dokážou lépe nastínit jak postup, tak možné komplikace při programování.

3.1. Boldi-vigna kódy

3.1.1. Názorný příklad zakódování a dekódovaní

Jak jsem již uvedl, vyjádření postupu se vzorci může být velmi matoucí, a proto je lepší postup přímo reálná ukázka. Základem je, že si určíme hodnotu *k*, což je faktor kódování. My si zvolíme *k=4* a číslo, které zakódujeme, bude *n=18*.

Jako první si vyhledáme, do jakého setu čísel, tvořícího se podle vzorce $[2^{hk}, 2^{(h+1)k} - 1]$, nám při faktoru *k* patří číslo *n*. Při procházení narazíme na sety $[1, 15]$ a $[16, 255]$, kde první je nevyhovující, proto hledáme set následující. Naše číslo *n* tedy zapadá do druhého setu a to pro nás znamená, že *h=1*, protože *h* se u setů inkrementuje od 0.

Dále již můžeme spočíst minimální binární kód podle rovnice $x = n - 2^{hk}$. Po dosazení dostáváme $x = 18 - 2^{1*4} = 2$. Poté zjišťujeme délku intervalu $z = 2^{(h+1)k} - 2^{hk} = 2^{(1+1)*4} - 2^{1*4} = 256 - 16 = 240$ a počet bitů $\log_2 z = \log_2 240 \sim 8$, respektive zaokrouhlujeme v celých číslech nahoru. Tyto výsledky dosadíme do nerovnice $x < 2^s - z$, tedy $2 < 2^8 - 240$, což je $2 < 17$ a tedy pravdivý výrok.

Pro nás to znamená, že prozatímní výsledek pro nás bude *x* na *s-1* bitech. Ať je číslo bitově delší, kratší nebo i záporné, budeme jej zobrazovat na prvních s-1 bitech. V našem případě tedy kód má velikost s-*1 = 7* bitů.

$0_9 0_8 0_7 0_6 0_5 0_4 0_3 1_2 0_1$ => V= 0000010

Nyní nám stačí dopsat unární zápis *h*, tedy $0^h 1$, což je nula zopakována jednou následovaná jedničkou. Výsledný kód tedy vypadá takto:

V= 010000010

Dekódování má následující postup. Na vstupu načítáme nuly, dokud nenarazíme na jedničku. Počet nul, které jsme do jedničky načetli je naše číslo *h*.

V= **01**0000010

Víme tedy, že *h=1,* a také dopředu víme, že *k=3*, což bylo předem dáno. Díky těmto informacím dokážeme spočítat, v jakém intervalu čísel se budeme pohybovat. Podle vzorce $[2^{hk}, 2^{(h+1)k} - 1]$ definujeme náš interval, tedy $[2^{1*4}, 2^{(1+1)4} - 1] = [16, 255]$. Jeho velikost $z = 2^{(h+1)k} - 2^{hk} = 2^{(1+1)*4} - 2^{1*4} = 256 - 16 = 240$. Z tohoto čísla se dozvíme počet bitů, a to pomocí $s = \log_2 z = \log_2 240 \sim 8$. Jelikož nevíme, zda je číslo zakódováno na *s* bitech, či na *s-1* bitech, načteme nejdříve *s-1* bitů a porovnáme, zda se jedno z čísel z intervalu v zakódované podobě rovná našemu sedmibitovému kódu. Jakmile najdeme shodu, což v našem případě jedno z prvních čísel intervalu, tedy *n=18* můžeme prohlásit naše hledané číslo. Kdybychom nenašli shodu ani v jednom čísle ze setu, načteme ještě jeden bit a zde shodu s jedním číslem určitě nalezneme.

3.1.2. Problémy při implementaci

Implementace tohoto kódu je vskutku jednoduchá, a díky známým vzorcům poměrně přímočará. Narážíme zde jen na málo problémů k řešení.

Problém s logaritmem

Je třeba správně ošetřit práci s logaritmy. Dají se poměrně lehce řešit bitovým posunem, avšak jelikož se pracuje s vysokými čísly, nastává problém posledních nejvyšších čísel. Při použití klasické jednoduché verze s posunem nastává problém, kdy počítané číslo v posledním intervalu skočí do minusových hodnot, a pokud člověk neotestuje správně tato čísla a neprojde si celý výpočet logaritmu, může lehce vytvořit fatální chybu.

Klasické řešení vypadá takto:

```
int boldiVigna :: GetLogg2(int i)
{
        int k = 1, j = 0;
        while (i>k) {
                k <<= 1;
                j++;
        }
        return j;
}
```

Aby se předešlo zmíněným problémům, je třeba přidat nezbytnou podmínku pro překročení hranice integeru.

```
int boldiVigna :: GetLogg2(int i)
{
        int k = 1, j = 0;
        while ((i>k)&&(k>0)) {
                k <<= 1;
                j++;
        }
        return j;
}
```

Problém s dekódováním

Při dekódování narážíme na velice nepříjemnou věc a to, že abychom zjistili, jaké číslo máme, musíme číslo testovat nejdříve na *s-1* bitech, a pokud jsme byli neúspěšní, opakujeme totéž testování na *s* bitech. Pro interval obsahující mnoho čísel je tento postup velice zdlouhavý a časově neefektivní.

3.2. Yamamotovy rekursivní kódy

3.2.1. Názorný příklad zakódování a dekódovaní

Vezměme si 2 příklady. Jeden pro *f=2* a jeden pro *f=3*. Pro jednoduchost a rozlišení si budeme popisovat všechny proměnné dolním indexem 2 a 3, a číslo, které se budeme snažit zakódovat, bude pro oba případy stejné a to *n=11* . Oddělovače nastavíme na $del_2=00$ a $del_3=111$.

První co vytvoříme, jsou naše nové řady čísel, které před prvním vyřazováním budou pro oba případy stejné. V druhém kroku vypustíme kombinace, které obsahují na začátku hodnotu z oddělovače (*viz. Tabulka 3.1*) a v třetím vyřadíme ty, jež mají menší délku než oddělovač a jejich hodnota je začátkem oddělovače (*viz. Tabulka 3.2*). Tímto postupem se dostaneme k naším zakódovaným řadám čísel pro naše zvolené oddělovače (*viz. Tabulka 3.3*).

n	K_n	del=00	del=111
1	0		
2	1		
3	00	X	
4	01		
5	10		
6	11		
7	000	X	
8	001	X	
9	010		
10	011		
11	100		
12	101		
13	110		
14	111		X
15	0000	X	
16	0001	X	
17	0010	X	
18	0011	X	
19	0100		
…	…		

Tabulka 3.1 ukazující odstranění hodnot obsahujících na začátku hodnotu oddělovače hodnota odpovídá začátku oddělovače

n	K_{n2}	del=00	K_{n3}	del=111
1	0	X	0	
2	1		1	X
3	01		00	
4	10		01	
5	11		10	
6	010		11	X
7	011		000	
8	100		001	
9	101		010	
10	110		011	
11	111		100	
12	0100		101	
13	0101		110	
14	0110		0000	
15	0111		0001	
16	1000		0010	
17	1001		0011	
18	1010		0100	
19	1011		0101	
…	…		…	

Tabulka 3.2 ukazující odstranění hodnot jejíž délka je menší než délka oddělovače a

n	K_{n2}	K_{n3}
1	1	0
2	01	00
3	10	01
4	11	10
5	010	000
6	011	001
7	100	010
8	101	011
9	110	100
10	111	101
11	0100	110

Tabulka 3.3 ukazující hodnoty pro čísla, které budeme využívat, zároveň barevné označení postupného tvoření kódu u ukázkových příkladů

V tuto chvíli budeme potřebovat pouze čísla do velikosti našeho čísla na zakódování, protože vyšší nevyužijeme.

Pro f_2 i pro f_3 pokračujeme následovně:

Nalezneme hodnotu v tabulce pro naše hledané číslo *n*, což je pro K_{n2} 0100 a pro K_{n3} 110 a vložíme si je před oddělovače. Výsledné hodnoty V_2 a V_3 pro nás budou mít prozatím tyto hodnoty:

$V_2 =$ **0100** | 00 $V_3 =$ **110** | 111

Nyní spočteme délky těchto nalezených hodnot, odečteme 1 a nalezneme k nim náležité kódy, které předsadíme před prozatímní vítězný kód. Délky jsou pro $0100_d = 4\text{-}1=3$ a pro $110_d = 3\text{-}1=2$, což směruje ke kódům 10 pro f_2 a 00 pro f_3.

$V_2 =$ **10** | 0100 | 00 $V_3 =$ **00** | 110 | 111

Nyní postup opakujeme. V obou případech je délka kódů 2, a tedy hledáme kódy pro hodnotu 1, kterou předsadíme před předešlý výsledek, čímž vytvoříme finální vzhled kódu, který bude mít tuto podobu:

$V_2 =$ **1** | 10 | 0100 | 00 $V_3 =$ **0** | 00 | 110 | 111

Jak je vidět u výsledku V_2 , kód v třetím bloku obsahuje „00“, což je i náš oddělovač, ale při ukázce dekódování pochopíme, že kód oddělovače na této pozici nemá vliv na úspěšnost dekódování.

Při dekódování budeme předstírat, že dostáváme bity ze streamu, který bude obsahovat 2x výsledek V_2, abychom viděli, že jde bez problémů odebrat jedno číslo bez načítání bitů z čísla následujícího.

Bity tedy budou seřazeny takto:

110010000110010000

Jakožto znalí tohoto kódování a při známém oddělovači je nám jasné, že první bit bude 1, kterou taktéž načteme, a s jistotou víme, že tento kód patří číslu *n=1* .

110010000110010000

Následně víme, že další skupinka bitů bude dlouhá *n+1* což je 2, takže načteme další 2 bity, tedy *10*. Při načítání dáváme pozor, zda začátek se náhodou nerovná již našemu oddělovači. Nyní si vytvoříme kombinace se stejnými pravidly jako při zakódování čísel (*viz Tabulka 3.3*) a vyhledáme, které číslo patří k tomuto kódu. Po nahlédnutí zjišťujeme, že naše hledané číslo je 3.

110010000110010000

Pokračujeme zase dále stejným principem. Zjistili jsme naše číslo *n=3* a tedy načítáme další 4 bity. Při načítání stále prohledáváme, zda se zde nenachází již oddělovač. Dostáváme *0100*, rovnající-se *n=11*.

1100100**00110010000**

Postup zůstává stále stejný, takže podle logiky bychom měli načíst dalších 12 bitů. Při načítání však narážíme na první bity *00*. A v tuto chvílí zastavujeme načítání a za dekódované číslo prohlásíme předešlé *n=11*.

1100100**00**110010000

3.2.2. Problémy při implementaci

Implementace Yamamotova rekurzivního kódu je těžší, než by se mohla zdát. Postup tvorby kódu je velmi pochopitelný a poměrně jednoduchý. Avšak při tvorbě kódů člověk narazí na pár potíží.

Problém číselných kombinací vysokých čísel

První problém přichází s kombinacemi nul a jedniček a vyřazování nevhodných variant. Člověk se může rozhodnout několika způsoby, jak toto provede, avšak žádný způsob nevede k elegantnímu řešení.

Vzhledem k tomu, že kód, který bude patřit k číslu, nelze lehce zjistit, díky odstraňování nevhodných variant, může vést k myšlence, že bychom před samotným užitím mohli všechny varianty propočítat a potom je následně uchovávat v paměti. Bohužel toto řešení by mohlo najít uplatnění pouze pro malá čísla, jelikož by takto vytvořená mohutná tabulka způsobila nečinnost programu již při inicializaci.

Další možností, zcela logicky opačnou než předchozí varianta, je vypočítávat kódy pro každé číslo znovu. Zde však extrémně narůstá časová náročnost a výpočet kódu jednoho čísla z horní hranice integeru by mohlo trvat i více jak několik hodin.

Asi nejlepší řešení by bylo, kdyby se našel kompromis mezi těmito dvěma extrémníma variantami. Jeden takový je např. si nechat ze začátku napočítat záchytné body a udržovat si je v paměti, a od kterých by se později dopočítávala hledané čísla. Tato varianta však zase musí udržovat relativně velkou tabulku kódů a také výpočet vyšších záchytných bodů přináší velkou časovou náročnost.

Z těchto důvodů bych doporučil nepoužívat toto kódování pro vysoké čísla.

Problém dekódování

Při dekódování nám přichází další problém, a to ten, že když dostaneme do rukou část kódu např. jako v ukázkovém příkladu 0100, který patřil číslu 11, musíme se k tomuto kódu dopočítat. Když vezmeme zhruba počet kombinací předcházejících této kombinaci, jedná se o

$$2^1 + 2^2 + 2^3 + 6 = 2 + 4 + 8 + 6 = 20\ možností$$

což je až dvojnásobný počet vygenerování kódů. Vzhledem k tomu, že vytváření těchto variant není taky moc jednoduchou akcí, a celé to kontrolování projde nespočtem podmínek, i když jen u kódů se stejným počtem bitů, není toto kódování přívětivé k rychlosti.

Když to shrneme, je třeba buď opět udržovat tabulku s kódy, což znamená vytvořit varianty dopředu, či je vytvářet právě při každém čísle, což sebou přináší časovou ztrátu. Zároveň je nutné mnoho podmínkami kontrolovat shody kódů a to pro každou skupinu bitů v kódu, a také kontrolování zda se zde nenachází oddělovač.

3.3. Tabu kódy

3.3.1. Názorný příklad zakódování a dekódovaní

Pro názorný příklad si zvolíme velikost bloků $m = 2$ a naše číslo k zakódování je $n = 14$. Jak bylo již napsáno, na výběru hodnoty oddělovače nezáleží, takže je nejlepší volba použít samé nuly- tudíž $DEL = 00$ o velikosti m.

Hodnoty, které budeme moct přiřadit v blocích, jsou následující (*viz. Tabulka 3.4*)

kód	účel
00	DEL
01	1
10	2
11	3

Tabulka 3.4 ukazuje definici využívaných kódů a čísel patřících k sobě k našemu příkladu

Dále pokračujeme výpočtem $g_m(k)$ a tím také určení hodnoty k:

$$g_m(k) = \frac{[(2^m-1)^k - 1](2^m-1)}{(2^m-2)} \qquad n \le g_m(k) - 1$$

$$g_2(1) = \frac{[(2^2-1)^1 - 1](2^2-1)}{(2^2-2)} = 3 \qquad 14 > 3-1 \qquad nevyhovuje$$

$$g_2(2) = \frac{[(2^2-1)^2 - 1](2^2-1)}{(2^2-2)} = 12 \qquad 14 > 12-1 \qquad nevyhovuje$$

$$g_2(3) = \frac{[(2^2-1)^3 - 1](2^2-1)}{(2^2-2)} = 39 \qquad \mathbf{14 \le 39-1} \qquad vyhovuje$$

Po výpočtu víme, že $g_2(3) = 39$ a tedy je $k=3$. Tyto hodnoty nás posouvají k vzorci na výpočet c.

$$c = n - g_m(k-1)$$

$$c = 14 - 12 = 2$$

A teď máme již vše k určení hodnot bloku b_i. Budeme se držet vzorce pro výpočet b a postupně tak stvoříme celý kód čísla, který bude mít tuto strukturu: $|B_2|B_1|B_0|DEL|$ s hodnotami z *Tabulky 3.4*.

$$bi = \left[\left\lfloor \frac{c}{(2^m-1)^i} \right\rfloor \% (2^m-1)\right] + 1$$

$$b_0 = \left[\left\lfloor \frac{2}{(2^2-1)^0} \right\rfloor \% (2^2-1)\right] + 1 = (2 \% 3) + 1 = \mathbf{3}$$

$$b_1 = \left[\left\lfloor \frac{2}{(2^2-1)^1} \right\rfloor \% (2^2-1)\right] + 1 = (0 \% 3) + 1 = \mathbf{1}$$

$$b_2 = \left[\left\lfloor \frac{2}{(2^2-1)^2} \right\rfloor \% (2^2-1)\right] + 1 = (0 \% 3) + 1 = \mathbf{1}$$

Po dosazení binárních hodnot do naší struktury dostáváme náš konečný kód |01|01|11|00|.

Při zpětném dekódování je nám dopředu známo *m = 2* a hodnota oddělovače *DEL = 00*. Na vstupu můžeme dostat stream několika čísel. Pro názorný příklad si zvolíme tento:

0101110001011100001011100

Jelikož je velikost bloků *m=2*, budeme načítat sety bitů po 2 bitech. Avšak víme, že první set nemůže být oddělovač, takže načítáme až od setu druhého a kontrolujeme.

01**01**1100010111000010111 00

0101**11**000101110001011100

<u>0101110</u>**00**0101110001011100

Jakmile jsme narazili na oddělovač, budeme se již věnovat jen bitům, které jsme do oddělovače načetli. Vytvoříme si opět tabulku hodnot (*Tabulka 3.4*) a přiřadíme hodnotám b_i jejich hodnoty.

$b_{0b} = 11$	$b_{1b} = 01$	$b_{2b} = 01$
$b_0 = 3$	$b_1 = 1$	$b_2 = 1$

Teď jen stačí vypočítat $g_m(k\text{-}1)$ a poté pomocí jedno vzorce postupně sečíst hodnoty do hodnoty výsledného čísla. Pro g_m uskutečníme stejné počty jako při zakódování a při tom nalezneme že *k=3* a proto nám bude ve vzorci vyhovovat *k=2*.

$$g_2(2) = \frac{[(2^2-1)^2 - 1](2^2-1)}{(2^2-2)} = 12$$

Pro výpočet čísla slouží vzorec:

$$n = g_m(k-1) + \sum_{i}^{k-1} (b_i - 1)\,(2^m - 1)^i$$

Což v důsledku znamená:

$$n = 12 + \ (b_0 - 1)(2^m - 1)^0 + (b_1 - 1)(2^m - 1)^1 + \ (b_2 - 1)(2^m - 1)^2$$

$$n = 12 + \ 2 + 0 + \ 0$$

$$n = \mathbf{14}$$

3.3.2. Problémy při implementaci

Implementace Tabu kódu je velmi jednoduchá, jelikož se skládá z pár vzorců, které jen stačí správně napsat. Jediný problém, který se zde může vyskytnout, je přehlédnutí a napsání vzorce špatně. Nachází se zde mnoho složitých rovnic s exponenty a různě ozávorkovanýma částmi.

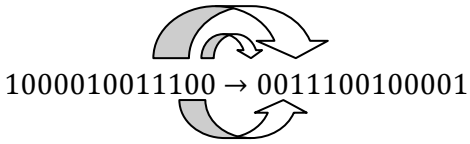
Dále je třeba dát pozor při implementaci vzorce pro výpočet *b*, kde se nachází dolní závora, tedy zaokrouhlení dolů. V tomto případě je třeba se ujistit, zda programovací jazyk a vybrané operátory zaokrouhlují dolů či nahoru nebo klasicky, jak jsme zvyklí z reálného života.

3.4. Wangovy flag kódy

3.4.1. Názorný příklad zakódování a dekódovaní

V tomto příkladě si ukážeme, jak zakódovat číslo n = 4252. Další informace, co budeme vědět dopředu, je velikost oddělovače, kterou si zvolíme 3.

Naše číslo n vlastní binární hodnotu 1000010011100. V prvním kroku musíme zapsat bity v opačném pořadí, tedy:



V dalším kroku procházíme sekvenci bitů a hledáme *f-1* nul, za které vkládáme jedničku, abychom zamezili výskyt oddělovače již v kódu.

$$0011100100001 \rightarrow 00111100110010011$$

Posledním nejlehčím krokem je dosadit za dosavadní kód oddělovač v plném znění. Výsledný kód bude:

$$001111001100100111000$$

Dekódování probíhá obzvlášť jednoduše. Prvním krokem je čtení bitů dokud nenarazíme na oddělovač. V tu chvíli si načtený blok bitů zpracujeme. Odstraníme oddělující 3 nuly a pokračujeme dále. Při znovu procházení bitů vyhledáváme *f-1* nul a vypouštíme následující jedničku.

$$00\cancel{1}11100\cancel{1}100\cancel{1}00\cancel{1}1 \rightarrow 0011100100001$$

Nakonec stačí bity zapsat v opačném pořadí a máme zde naše hledané n.

$$0011100100001 \rightarrow 1000010011100 = 4252$$

3.5. Yamamoto flag kódy

3.5.1. Názorný příklad zakódování a dekódovaní

K tomuto příkladu si schválně vezmeme takové číslo, abychom viděli, jaké všechny situace mohou nastat při zakódování. Jak se uvidí, nebude třeba dlouhého vysvětlování, jelikož logika, která sice je složitější než v případě Wangova kódu, je i tak velmi jednoduchá.

Takže naše číslo *n*, se kterým budeme pracovat, je 1736, jehož binární hodnota je 11011001000. Určíme oddělovač *f*, který si pro tento příklad vybereme 3bitový o binární hodnotě 100.

Začínáme tedy s postupem. Jako první věc, co musíme udělat je odstranit první bit čísla *n*, o kterém víme, že zde je zbytečný, jelikož všechny čísla začínají tímto bitem.

$$n = \cancel{1}1011001000$$

Projdeme postupně všechny bity, a budeme hledat, kde se v kódu čísla nachází první 2 bity oddělovače tedy 11. Procházení a vyhledávání by se dalo znázornit tímto postupem:

$$\begin{aligned} n = \quad & 1011001000 \\ & 10 \\ & \;\;10 \\ & \;\;\;10 \\ & \;\;\;\;1 \\ & \;\;\;\;\;10 \\ & \;\;\;\;\;\;1 \\ & \;\;\;\;\;\;\;1 \end{aligned}$$

Nyní si tento postup projdeme slovně. Když začneme, zkoušíme hledat shodu od prvního bitu zleva našeho zkráceného čísla. Shody se nám dostává u prvního bitu oddělovače, prozkoumáme tedy druhý bit. Zde shodu nacházíme také, proto za tyto bity následně vložíme negaci konce oddělovače.

Jelikož jsme nalezli shodu s části oddělovače, pokračujeme dále až od třetího bitu čísla. Shodu opět nalézáme, avšak shoda s druhým bitem oddělovačem není nalezena. Po takovéto situaci pokračujeme normálně dále a zkoumáme následující čtvrtý bit, zda právě v něm nezačíná kód oddělovače. Opět nastává ta samá situace jako u první dvojice, shoda s prvním bitem a také shoda v následujícím bitu. Proto za tyto bity také vložíme negaci konce oddělovače.

Jelikož byl nalezen kandidát na oddělovač, pokračujeme až bitem šestým. Ten se však neshoduje. Jdeme tedy prozkoumat rovnou bit sedmý, kde se shoda nalézá. Shoda je nalezena tak samo u osmého s druhým bitem oddělovače. Za tuto dvojici tedy taktéž vložíme negaci posledního bitu oddělovače.

Když procházíme tímto způsobem pole bitů dál, zjišťujeme, že shodu již nikde nenalézáme, a tak nakonec, jako vždy u kódů s postfixovým oddělovačem, vložíme hodnotu oddělovače na konec celého oddělovače. Výsledný kód po všech těchto bude vypadat takto:

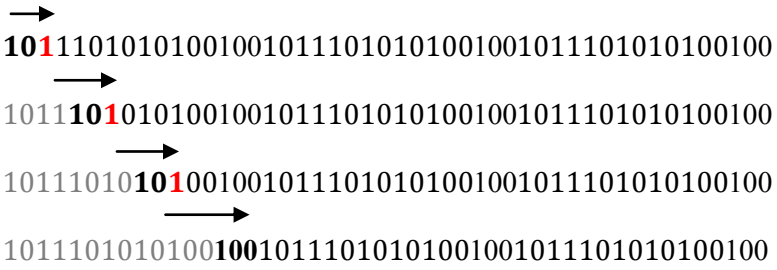
11011001000 → ~~1~~1011001000 → **10**1**11**0**10**1**0**0 → **1011101010100**100

Dekódování Yamamotova flag kódu je jednodušší, než jej zakódovat. Při dekódování známe akorát hodnotu oddělovače *f* = 100 a i jen tohle nám postačí.

Při načítání bitů načteme tolik bitů, dokud nenarazíme na náš oddělovač. Pokud si představíme stream bitů, může to vypadat nějak takto:

10111010101001001011101010100100101110101010100100

Důležité však je nalézt správný oddělovač. Ten správný poznáme tak, že budeme vyhledávat i oddělovače s negovanou poslední hodnotou. Při jejich nalezení, jejich kód nebudeme brát v potaz u hledání pravého oddělovače. Takže při hledání ve streamu vyhledáváme oddělovač takto:



Jakmile nalézáme oddělovač, oddělíme jej a ponecháme si set bitů, které jsme načetli. Tyto bity následně analyzujeme. Nejdříve projdeme bit po bitu a pokusíme se nalézt kód oddělovače bez posledního bitu. Po identifikaci tohoto kódu, vypustíme následující bit.

10+1101010100

10+110+010100

10+110+010+00

10+110+010+00 → **1011001000**

Až projedeme celý kód, vložíme před něj jedničku a máme tím naše hledané číslo.

10+110+010+00 → **1**1011001000 → **1736**

3.5.2. Problémy při implementaci

Implementace tohoto kódu je více než jednoduchá, ale i přesto je třeba si dát pozor na oddělovač, který tvoří některá omezení.

Problémy s oddělovačem

Jak už bylo řečeno, oddělovač může být o minimální délce 3. Je to tak, protože délka 1 je nereálná, a při délce 2 bitů by prakticky každé druhé číslo mělo přiložený bit za sebou, což by se podepsalo znatelně na délce kódu.

Avšak oddělovač skýtá mnoho problémů. Hodnota oddělovače je již omezená vzorci, jelikož bylo dokázáno, že tyto právě nedokážou v kódu vytvořit neřešitelné klamání falešným oddělovačem. Dále je pak na lidech aby dokázali nalézt ty, které řešitelné jsou. Jeden takový je v příkladě výše, kde hodnota oddělovače 100 se objevuje i před oddělovačem.

10111010101001**00**

To však jde ošetřit tím, že buď budeme kontrolovat, zda část domnělého oddělovače není negovaným koncem oddělovače, jež byl vložen kvůli shody při zakódování a nebo hned při načítání budeme odstraňovat tyto negované části oddělovače, čímž se vyhneme sice druhému hledání těchto negací, avšak to přinese větší zamyšlení k implementaci a složitější řešení.

4. Testování efektivity komprese

Testování probíhá na souborech dat, každý o velikosti miliónu náhodných čísel. 4 soubory obsahují po osmi bitech celé spektrum kladného spektra integeru. Další soubor obsahuje malé čísla do 5 bitů a poslední náhodné čísla z celého spektra čísel, kde pravděpodobnost vysokých čísel je mizivá, tedy se jedná spíše o nízká čísla (viz *Tabulka 4.1*).

Typ dat	rozsah	poznámka
0-8 bitů	(0, 8)	data jsou náhodná s rovnoměrným rozložením
8-16 bitů	< 8, 16)	
16-24 bitů	< 16, 24)	
24-32 bitů	< 24, 32)	
0-5 bitů	(0, 6)	
normální	(0, 32)	funkce normálního rozložení $f(x) = (2*pi*s^2) -0.5* \exp(-0.5*s -2*(x-m)^2)$ kde bylo použito M=0 a S=128

Tabulka 4.1 definující, jaká data se nachází v testovacích souborech

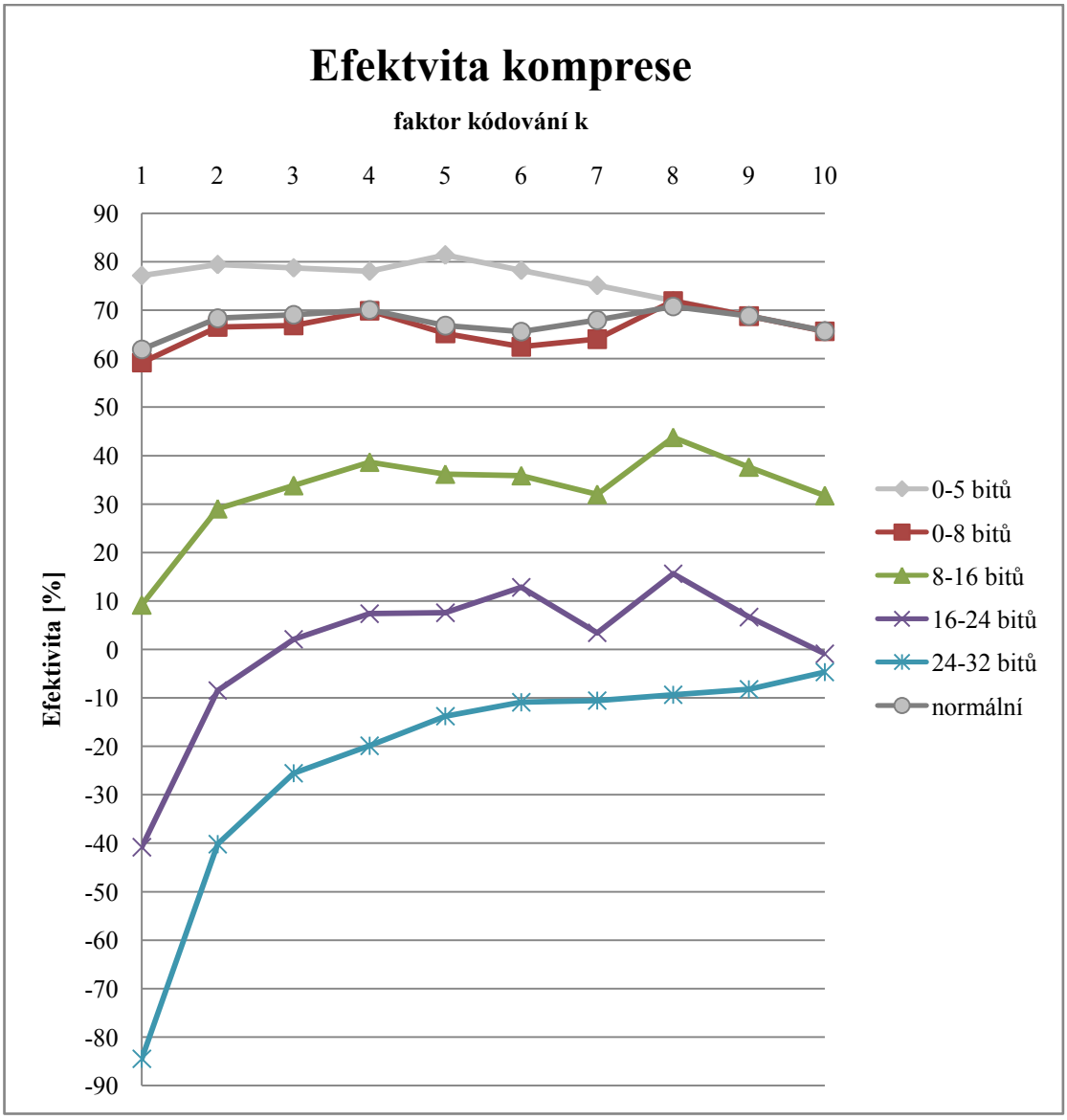
4.1. Testování efektivit variant kódů

Prakticky všechny kódy mají několik svých variant a ne všechny jsou pro kompresi ideální. Varianty můžou být různého typu. Pro kódy s postfixovým oddělovačem varianty vznikají velikostí oddělovače a většinou také kombinací nul a jedniček v oddělovači. U kódů s výpočtem kódu zase varianty vytváří změna hodnoty proměnné. Pokud chceme zjistit efektivitu kódů a popř. porovnat je mezi sebou, musíme nalézt jejich nejefektivnější varianty.

4.1.1. Boldi-vigna kódy

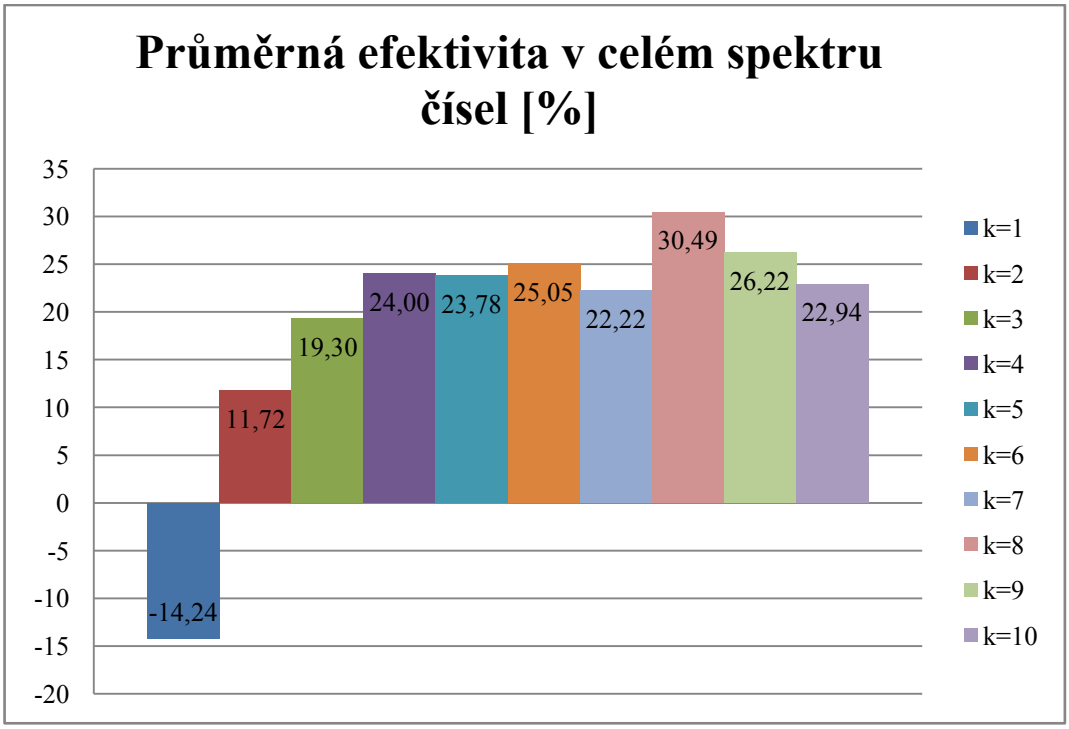
Při pohledu na graf (*Graf 4.1*) zjišťujeme, že až na výjimky, různé hodnoty faktorů mají různé výsledky v jiném spektru čísel. Při kompresi nejmenších čísel je z čeho vybírat, dalo by se říct, že faktory dosahují do velikosti 6 poměrně vyrovnaných výsledků. Pokud bychom však chtěli brát v potaz větší čísla, měli bychom se spíše soustředit na faktory 4 a 8.

Ovšem při kompresi těch největších čísel zjišťujeme, že je komprese kontraproduktivní, zejména u faktorů s nízkou hodnotou. Záleží tedy, jaké čísla budeme používat nejčastěji a podle toho by se měl vybrat příslušný faktor, zpracovávající určité hodnoty nejkvalitněji.



Graf 4.1 zobrazující efektivitu komprese u Boldi-Vigna kódu na všech souborech dat s faktorem kódování k=1 až k=10

Pokud však víme, že naše číselné spektrum bude obsahovat čísla opravdu náhodné, měli bychom se řídit podle grafu s průměrnými hodnotami, kde jednoznačně prokazuje sílu komprese faktor s hodnotou 8(*viz Graf 4.2*).



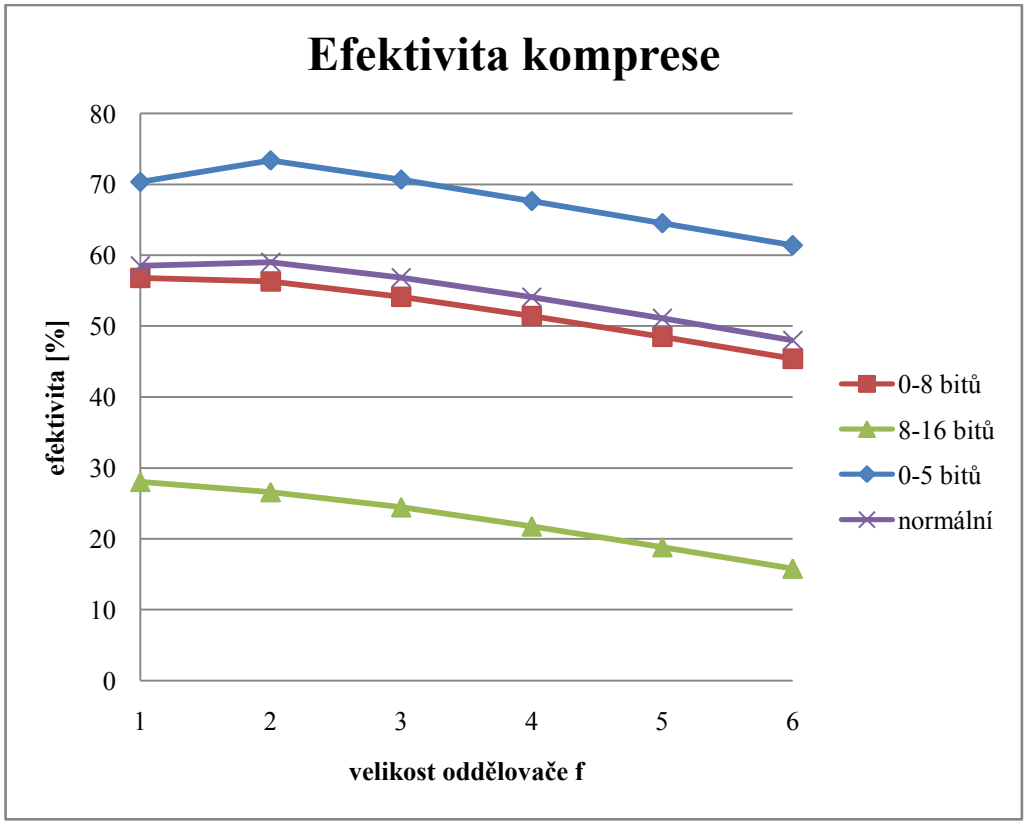
Graf 4.2 zobrazující celkovou úspěšnost, tedy průměr ze 4 souborů dat mapující celé spektrum čísel, Boldi-Vigna kódu s faktorem k=1 až k=10

4.1.2. Yamamotovy rekursivní kódy

Jak již bylo napsáno dříve, nejpoužívanější možnosti jsou zde při užití oddělovače délky 2 a 3. To již svědčí o efektivitě, ale není na škodu si dokázat proč.

Ozkoušeny byly varianty pro oddělovače o velikosti 1-6 bitů. Ostatní možnosti oddělovačů nebylo třeba zkoušet, jelikož by díky mohutnému oddělovači působili značně kontraproduktivně.

Testy prokázaly obavy z velké časové náročnosti při velkých číslech. Proto bereme čísla více jak 16 bitová za nevhodná pro zakódování a po domluvě s vedoucím panem Platošem nebudou vyhodnoceny.

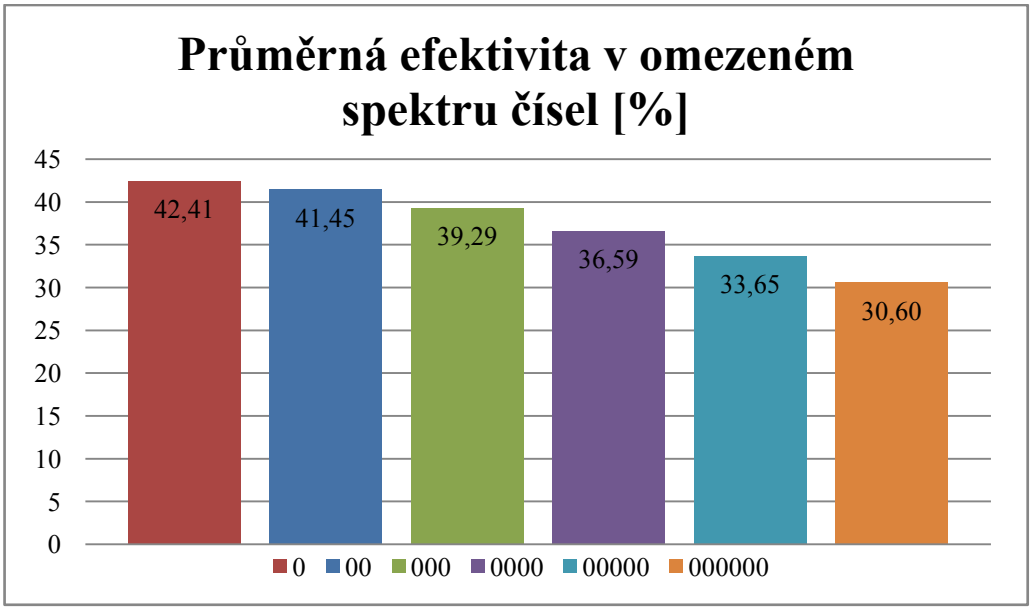


Graf 4.3 zobrazující efektivitu komprese u Yamamotova rekurzivního kódu na některých souborech dat s velikostí oddělovače 1 – 6 bitů

Jak vidíme na grafu (viz *Graf 4.3*), velké čísla nemají problémy jen s časovou náročností, ale také s efektivitou. Když pohlédneme na to, že druhý set čísel v nejlepších případech ani nedosahuje 30% úspory, můžeme si domyslet, jaká účinnost by byla u dalších 2 setů, a je velmi pravděpodobné, že by klesla do záporných hodnot, což by v důsledku znamenalo, že by zabíraly více jak 32 bitů. Byla by tedy mohutnější a zároveň by trvalo velmi dlouho, než by se zjistilo, o jaké číslo se jedná.

Když na graf koukneme z pohledu velikosti oddělovače, zjistíme, že nejvíce účinné jsou oddělovače s velikostí 1-3 bity (*viz Graf 4.4*) a s větší velikostí účinnost permanentně klesá. Otázkou zůstává, proč se v praxi více používá 3 bitový oddělovač než 1 bitový, i když má, sice o minimum, horší výsledky. Opodstatnění je asi v celkové náročnosti tvorby kódu, kdy se kódování použije pouze na malá čísla, kde profituje oddělovač o velikosti 3.

Pokud opět pohlédneme na celkové zhodnocení (*Graf 4.4*), můžeme trochu polemizovat o celkové účinnosti. Jak jsem již naznačil, další testované sety čísel by končily nejspíš v záporných hodnotách. Pokud vezmeme v potaz, že průměrné hodnoty u nejmenších čísel dosahují maximálně 42% úspory, což je jen o kousek lepší než nejlepší hodnoty u jiných kódování, avšak z celého spektra čísel, můžeme se domnívat, že celková efektivita Yamamotového rekurzivního kódu by s velkou pravděpodobností klesala daleko pod 20%.

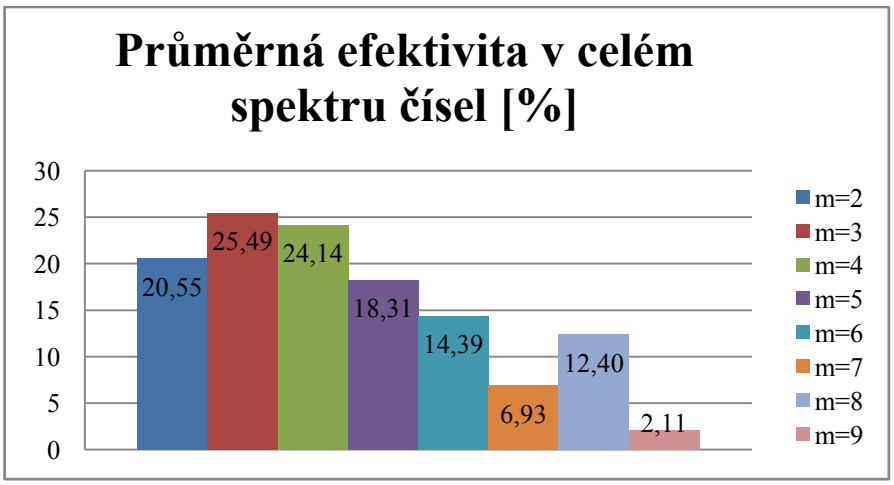


Graf 4.4 zobrazující průměrnou efektivitu komprese u Yamamotova rekurzivního kódu na 2 souborech dat, obsahující čísla z první poloviny celého spektra při velikosti oddělovače 1 – 6 bitů

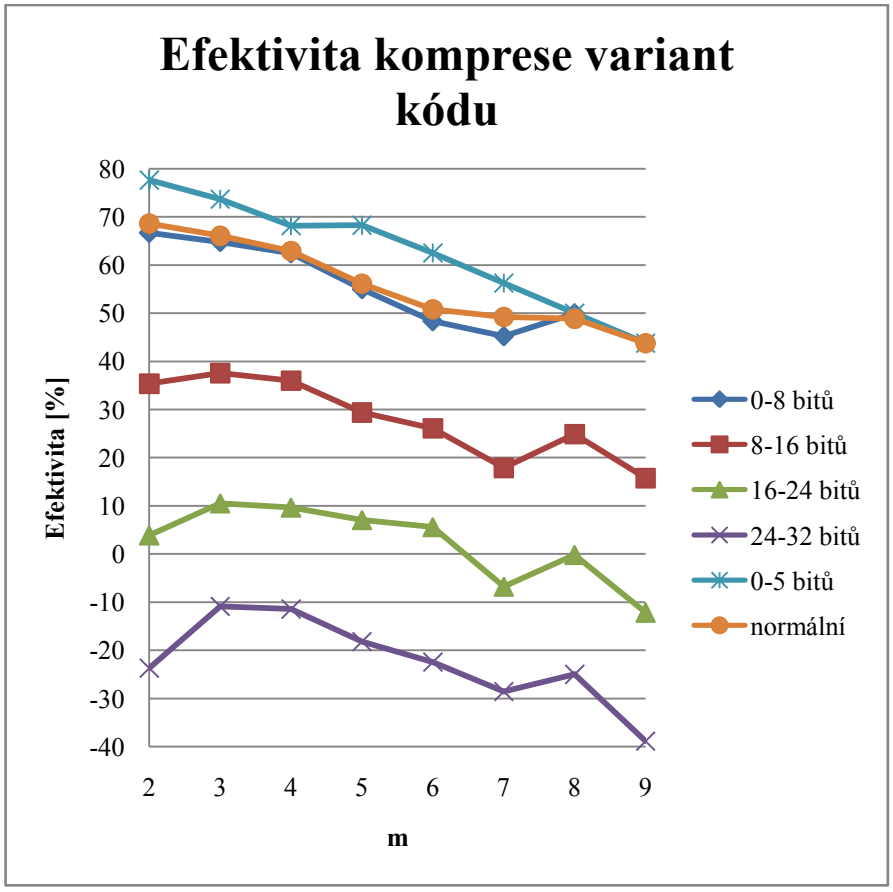
4.1.3. Tabu kódy

Tabu blokový kód má 2 důležité vlastnosti. Tím že je blokový přináší do dekódování elegantnost v rychlosti nalezení oddělovače, a co se tvorby kódu týče, jistým způsobem patří mezi ty rychlejší. Tohle však má svůj negativní efekt. Oproti jiným kódům, zde má početní skupina čísel předdefinovanou délku. Pro vyšší čísla ze skupiny je délka akorát, avšak u nižších nacházíme zbytečné bity. Tento nežádoucí efekt se prohlubuje s větší délkou oddělovače.

Co se týče už přímo efektivity variant kódů, při pohledu na *Graf 4.6* zjišťujeme, že na nejnižší čísla je nejefektivnější nejkratší oddělovač *m=2*. Zato na další čísla už výsledky vychází podle stejné režie, kdy oddělovače o velikostech 3 a 4 kralují efektivitě a všechny ostatní poměrně značně ztrácí. To se promítá i na průměrné hodnotě z celkového spektra čísel (*Graf 4.5*), kde jde od prvního pohledu poznat, jak moc jsou velikosti *m* 3 a 4 účinné oproti ostatním.



Graf 4.5 zobrazující celkovou efektivitu komprese Tabu kódu na souborech dat, mapujících celé spektrum čísel, s velikostí bloku 2 – 9 bitů



Graf 4.6 zobrazující efektivitu komprese Tabu kódu na všech souborech dat s velikostí bloku 2 – 9 bitů

4.1.4. Wang flag kódy

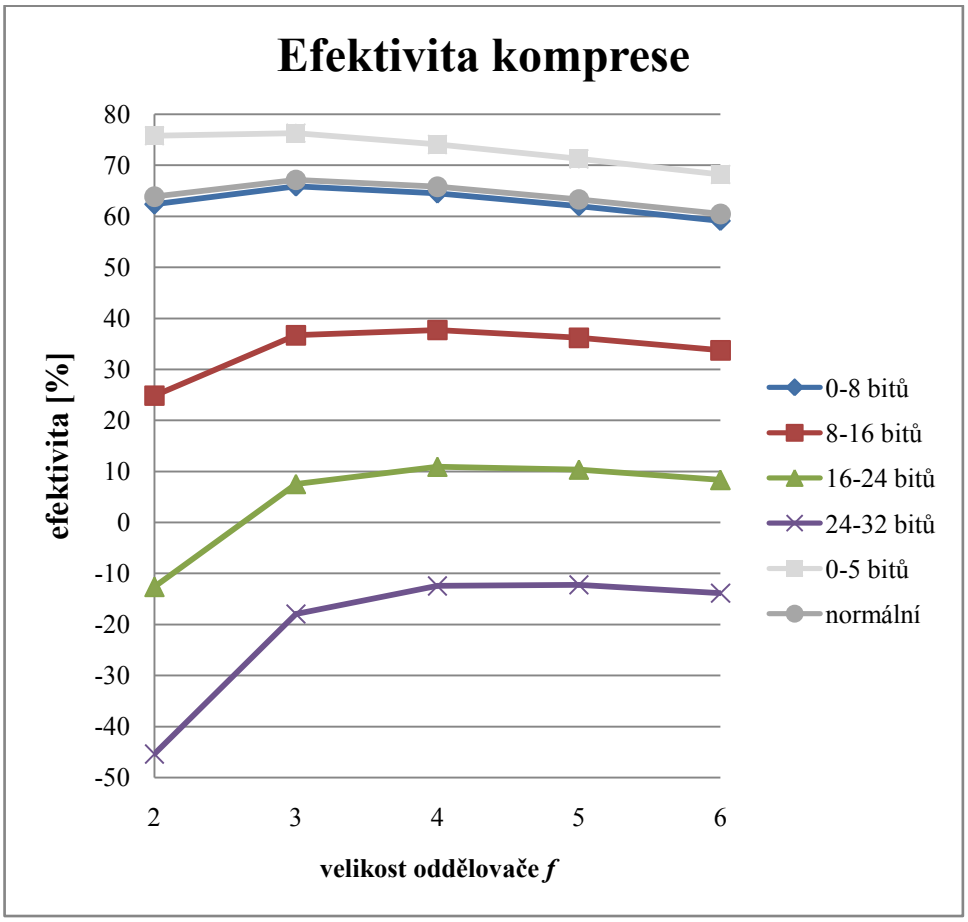
Tvorba Wang flag kódů je enormně jednoduchou záležitostí. Otázkou je jak velký vliv na kód bude mít velikost oddělovače. Při zamyšlení docházíme k logickým souvislostem mezi velikostí oddělovače a velikostí výsledného kódu.

Když zvolíme oddělovač krátký, u malých čísel tak docílíme malé velikosti celkového kódu, avšak při větších číslech velikost rapidně naroste, jelikož zde přichází větší šance, že narazíme na situace, kdy budeme muset vkládat do kódu jedničky pro eliminaci nevhodných kandidátů na oddělovač.

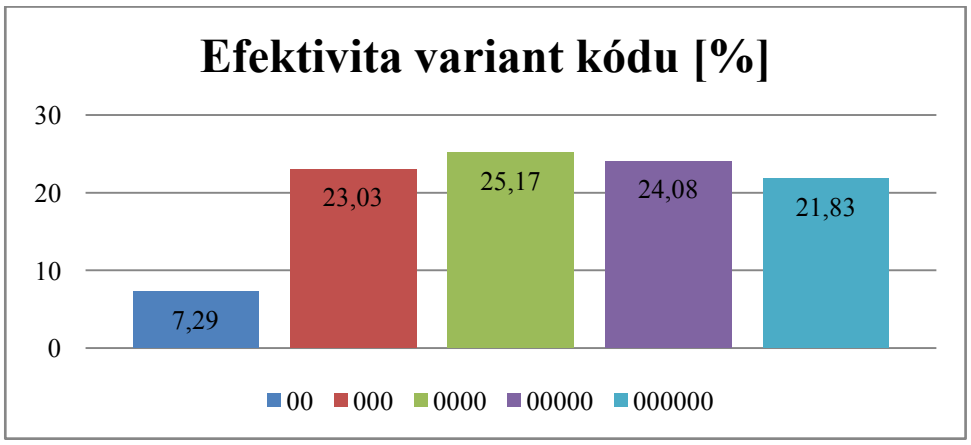
Když naopak zvolíme oddělovač velký, šance na to, aby se v kódu objevil kód oddělovače, je mnohem menší. To nahrává vysokým číslům, která takto ušetří mnoho bitů oproti kódům s kratšími oddělovači. To však nemůžeme říci o využití u malých čísel, kdy sice nalézt shodu s oddělovačem v kódu je minimální, ale zato kód získá robustnost díky dlouhému oddělovači.

Je tedy jasné, že pokud je třeba komprimovat data z celého spektra, je třeba jít zlatou střední cestou. Při pohledu na *Graf 4.7* zjišťujeme, že použití oddělovače o velikosti 2, je zbytečné, jelikož rozumných výsledků dosahuje jen při nejmenších číslech a na větších výrazně pokulhává. Další velikosti se však jeví celkem vyrovnaně a podporují myšlenku ze začátku o důsledku velikosti oddělovače na výsledný kód. Pokud tedy bychom předpokládali, že budeme pracovat s čísly hlavně z určité škatulky, můžeme si vybrat, který oddělovač tomu zrovna vyhovuje nejvíce.

Pokud však chceme používat celé spektrum čísel, při prozkoumání *Grafu 4.8* zjistíme, že nejefektivnější je oddělovač o velikosti 4.



Graf 4.7 zobrazující efektivitu komprese Wang flag kódu na všech souborech dat s velikostí oddělovače 2 – 6 bitů

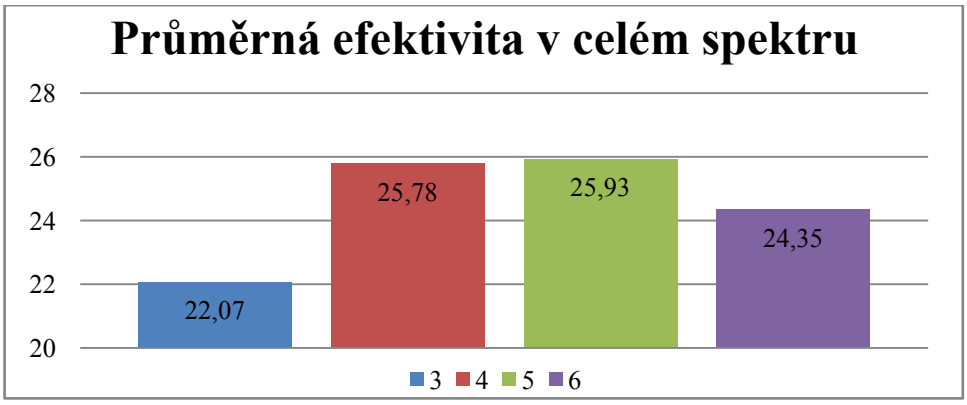


Graf 4.8 zobrazující celkovou efektivitu komprese Wang flag kódu na souborech dat, mapujících celé spektrum čísel, s velikostí oddělovače 2 – 6 bitů

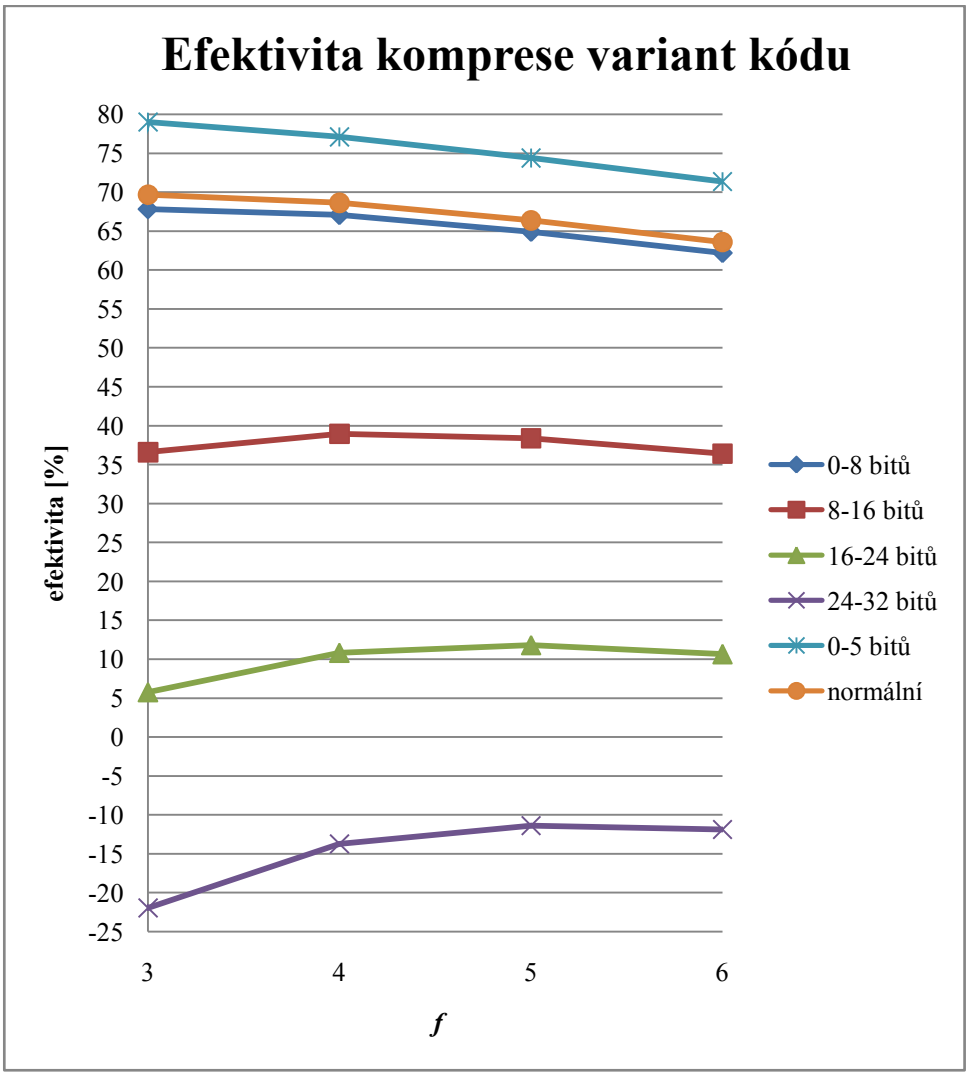
4.1.5. Yamamotovy flag kódy

Yamamotovy Flag kódy jsou vylepšené Wang Flag kódy. Platí zde podobné úvahy o tvorbě kódu, akorát se složitější logikou avšak rychlejším řešením. Vliv na úspěšnost komprese by měly mít

prakticky taky lepší díky ušetření bitu na každém čísle. Při pohledu na graf celkové efektivity Yamamotového kódu (*Graf 4.9*) a Wang kódu (*Graf 4.8*) zjišťujeme, že kód nabral nepatrné zlepšení, tudíž se můžeme domnívat, že bude výhodnější pro využití použít kód Yamamotův.



Graf 4.9 zobrazující celkovou efektivitu komprese Yamamotova flag kódu na souborech dat, mapujících celé spektrum čísel, s velikostí oddělovače 3 – 6 bitů



Graf 4.10 zobrazující efektivitu komprese Yamamotova flag kódu na všech souborech dat s velikostí oddělovače 3 – 6 bitů

Stejně jako ve většině případů kódů s oddělovači, platí zde stejné pravidlo. Kód malých čísel nabývá nejlepších výsledků u oddělovačů s malou délkou a postupně tato účinnost přechází u větších čísel na delší oddělovače (což můžeme pozorovat na *Grafu 4.10*). Pokud se budeme pohybovat v číslech o velikosti integeru, můžeme usoudit podle toho, že oddělovač s velikostí 6 nikdy nepředčil oddělovač s velikostí 5, tak velikosti 6 a více budou pro nás nedůležité a zbytečné. Dále můžeme usoudit, že pokud budeme využívat celé spektrum čísel, oddělovač o velikosti 3 využijeme tehdy, kdy budeme vědět, že se budou čísla objevovat výhradně z nejnižšího bloku čísel.

4.2. Celkové srovnání kódů

Po tom co jsem důkladně probral určitá kódování s jejich variantami, je na čase na celkové zhodnocení a porovnání nejlepších variant kódů mezi sebou. Dále určitě bude zajímavé, které kódování se bude nejvíc hodit na celkové spektrum čísel, i na jednotlivé části spektra (*viz. Tabulka 4.1*).

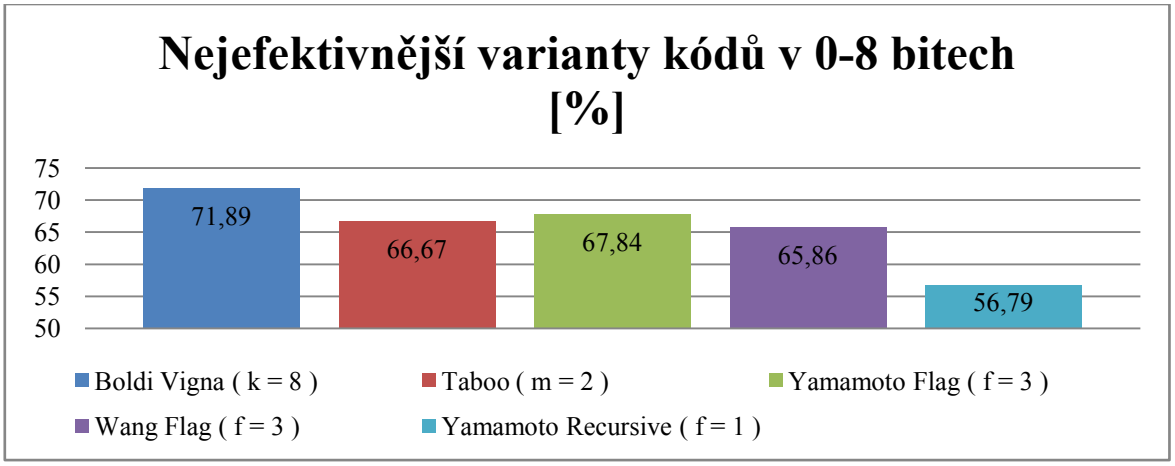
4.2.1. Kódy v testu 0-8 bitů

Testování na datech 0-8 bitů, tedy na číslech od 1 do 255, je jeden z nejdůležitějších testů vůbec, jelikož se tyto hodnoty v reálném životě využijí nejvíce.

Pokud pohlédneme na graf (*Graf 4.11)*, zjistíme, že v tomto spektru čísel s velkým náskokem vítězí kód Boldi-vigna. Rozdíl 4% sice nevypadá jako velký, avšak při ohromném množství dat, kvůli kterému se komprese vyvíjela, to může znamenat velké přilepšení. Naopak Yamamotův rekurzivní kód v jeho nejlepší variantě potvrzuje pouze poloviční úsporu. Ostatní kódy si vedou vyrovnaně a dalo by se říct dobře.

Zajímavé však je, že Boldi-Vigna kódy i ve variantách o $k = 4$ a $k = 9$ dosahují lepších výsledků než zde na grafu Yamamotův Flag kód, a dokonce další 4 varianty Boldi-Vigna podobných výsledků. Svědčí to o velkém potenciálu Boldi-Vigna kódů na malých číslech.

Další zajímavostí je, že zde ukázané kódy Boldi-Vigna, Taboo a Yamamotův rekurzivní mají nejlepší výsledky v tomto spektru, tak i v celkovém spektru čísel.

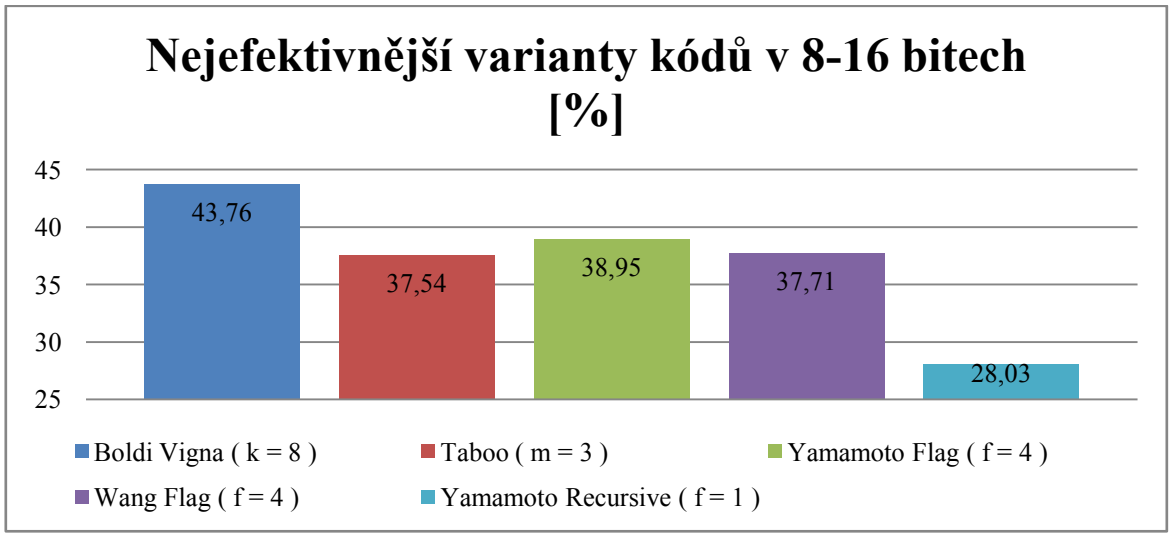


Graf 4.11 zobrazující nejefektivnější zástupce kódů pro datový soubor 0-8bitových čísel

4.2.2. Kódy v testu 8-16 bitů

Testování dat se týká čísel v rozmezí 256 až 65 535. Tyto čísla jsou také velmi používaná, ne však už tolik jak předchozí set.

Při zkoumání grafu (*Graf 4.12)* zjišťujeme, že Boldi-Vigna opět převyšuje ostatní kódy a prakticky se nenachází rozdíl v tom, jak dopadly nejlepší varianty mezi sebou. Rozdíl se však nachází v použitých parametrech, kdy u prostředních 3 kódů byl změněn. Tudíž musíme počítat s tím, že si minulé varianty nevedou stále tak dobře. Každopádně Boldi-Vigna kód stále s faktorem kódování $k = 8$ dokazuje jeho velikou efektivitu v dalším spektru. Varianty s $k = 4$ a $k = 9$ již nedosahují lepších výsledky než ostatní kódy, ale stále dosahuje alespoň vyrovnaných výsledků.

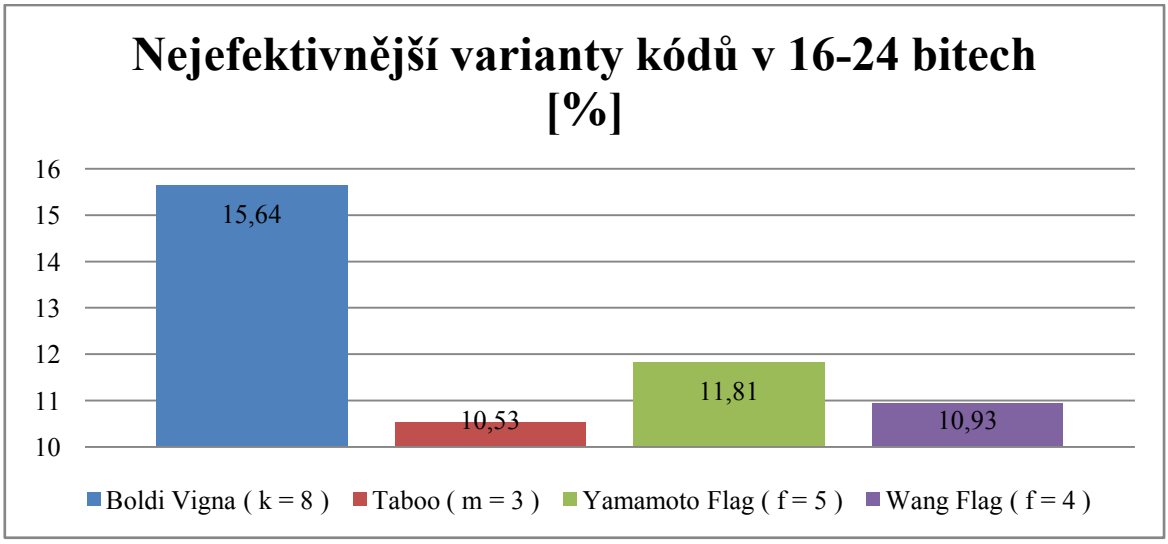


Graf 4.12 zobrazující nejefektivnější zástupce kódů pro datový soubor 8-16bitových čísel

4.2.3. Kódy v testu 16 - 24 bitů

Test probíhal na datech z číselného spektra mezi 65 536 až 16 777 215. Jak je vidět čísla stále patří mezi často užívaná a avšak o něco méně než předešlé sety.

Zde již nešlo testovat Yamamotův rekurzivní kód, avšak stejně podle mých domněnek by se pohyboval kolem 0% úspory, ne-li záporné. Jak vidíme na *grafu 4.13*, kódy stále zaujímají stejné pozice jako v předešlých setech. Co se týče Boldi-Vigna kódu, již v třetím setu čísel zaujímá první pozici a to se stále stejným faktorem kódování $k = 8$. Dříve zmíněné faktory se již propadají daleko pod ostatní kódování, čímž zesilují efekt, že $k = 8$ je unikátní.

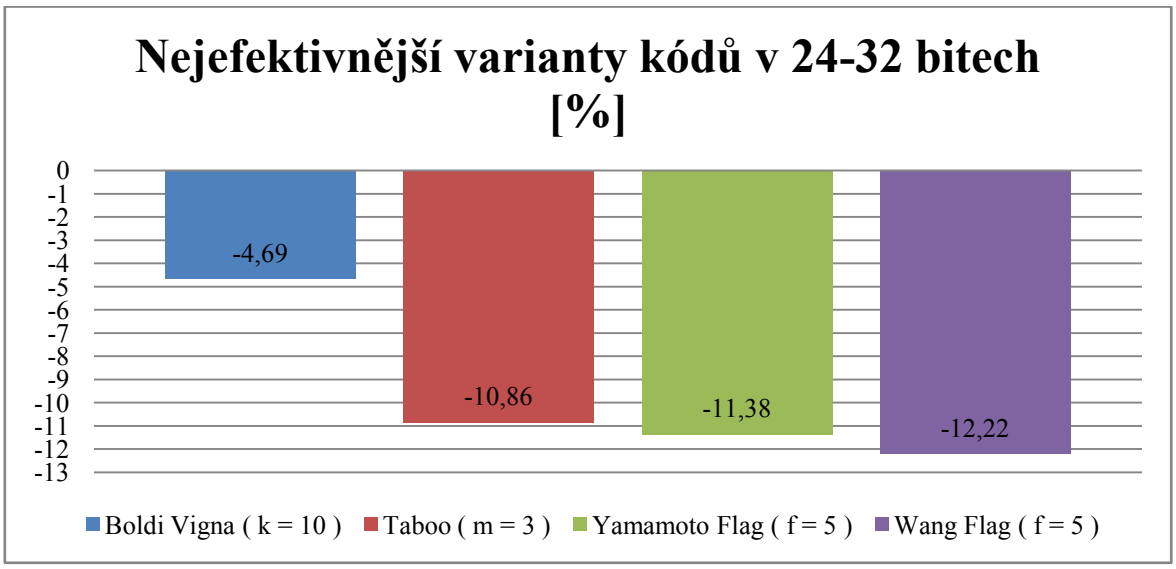


Graf 4.13 zobrazující nejefektivnější zástupce kódů pro datový soubor 16-24bitových čísel

4.2.4. Kódy v testu 24 - 32 bitů

Testy na datech probíhají s čísly od 16 777 215 do 2 147 483 647. Tyto čísla už jsou používaná značně méně než předchozí sety, ale občas je jich třeba využít. Pokud se zamyslíme nad těmi nejvyššími čísly, tak nám může být jasné, že právě tyto budou spíše velmi ztrátové než úsporné, jelikož jejich 32bitovou délku přesáhneme už připsáním oddělovače. Opět zde vynechávám Yamamotův rekurzivní kód, kvůli velké časové náročnosti.

Pokud pohlédneme na graf (*Graf 4.14)*, vidíme že Boldi-Vigna kód je z daleka nejefektivnějším kódem v našem výběru a to v každém setu spektra čísel. V tomto rozmezí sice kraluje *k=10*, avšak *k=8* stále dosahuje lepších výsledků než nejlepší zde kandidáti jiných kódů. Co se týče ostatních kódů. Tabu kód ukázal větší efektivitu oproti zbylým dvěma kódům a oproti minulému setu, kde si vedl nejhůř, si zde vede nejlépe.



Graf 4.14 zobrazující nejefektivnější zástupce kódů pro datový soubor 24-32bitových čísel

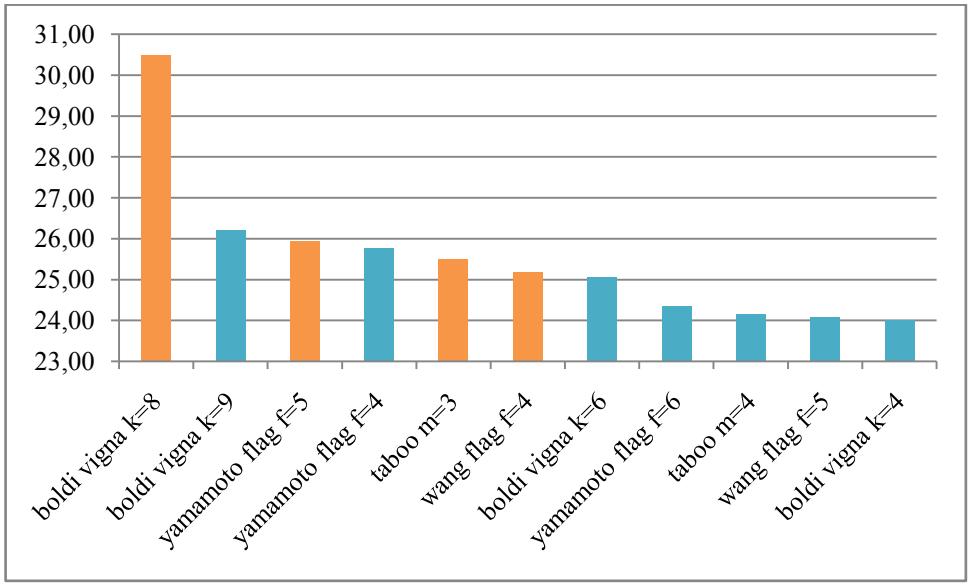
4.2.5. Kódy v testu celého spektra

Nejedná se o nový test, ale zprůměrování předešlých čtyř setů, tvořící kladné spektrum integeru.

Když se poohlédneme zpět, tak Boldi-Vigna kódy dominovali každému setu čísel a to dokonce několika variantami. Zatímco Yamamotův rekurzivní kód poměrně ztrácel a v některých případech ani nešel naměřit. Proto v *Grafu 4.15* použijeme nejúspěšnější kódy a varianty kódů, i když zástupce tohoto kódu se tam již bude nacházet. Zato vypustíme Yamamotův rekurzivní, protože se do této statistiky kvůli chybějícím datům nedá zařadit.

Při pohledu na *Graf 4.15* vidíme, jak moc velký rozdíl je mezi Boldi-Vigna s *k=8*, který s přehledem vedl statistiky prvních 3 ze 4 sektorů před všemi kódy a jejich variantami. Konstrukcí kódu ostatní převyšuje a mezi 11 nejúspornějšími kódy má 4.

Co se týče Wang a Yamamotova flag kódů, jde zde vidět, že Yamamotovi se vylepšení Wangova kódu povedlo a dokázal jej nepatrně zlepšit. Zlepšení však mělo jít poznat hlavně na čase.

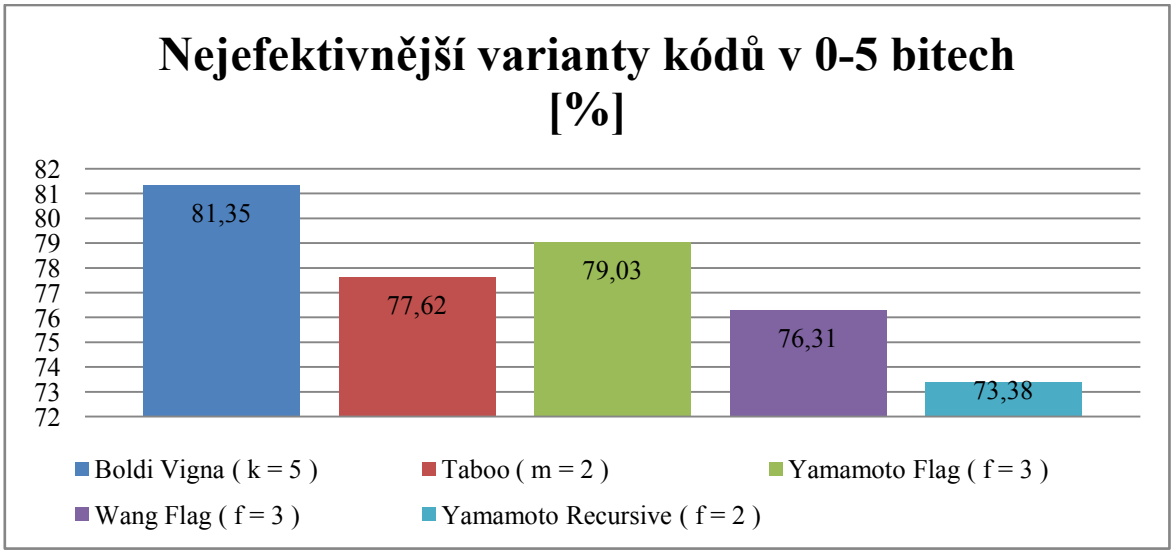


Graf 4.15 zobrazující k porovnání celkově nejefektivnější varianty všech kódů, s barevným vyznačením nejefektivnějších zástupců kódů

4.2.6. Kódy v testu 0-5 bitů a normálních čísel

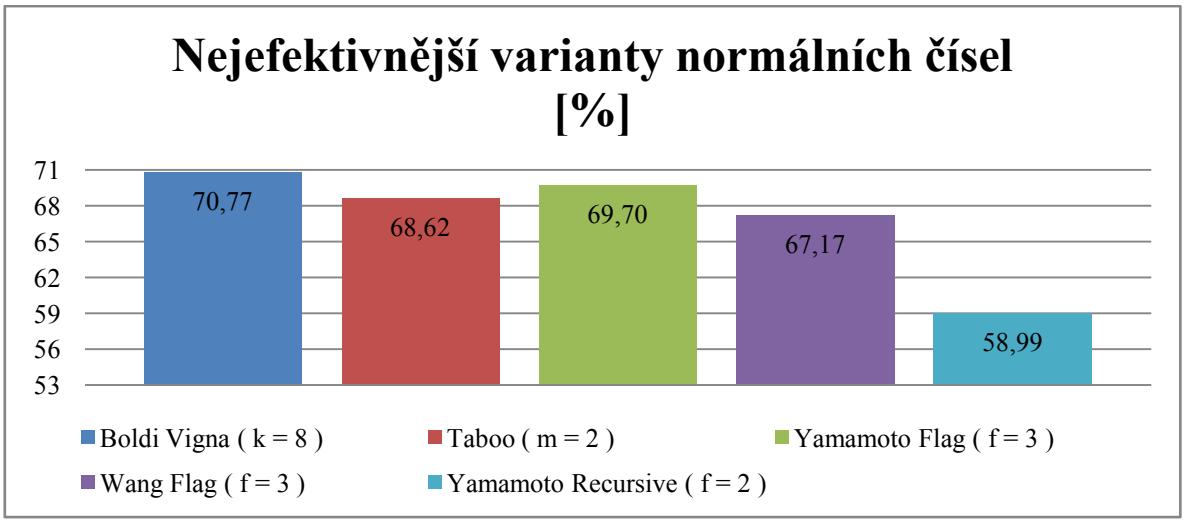
Tento test se týká výhradně extrémně malých čísel. Často se stává, že používáme pouze jednotky, či trošku více. Proto je dobré otestovat kódy i na malá čísla v rozsahu 1 až 32. Další speciální případ je použití čísel nízkých, ale zároveň i vyšších, které se výjimečně mohou objevit. Proto budeme testovat i čísla ze setu nazvaného „Normální“.

Jak jde vidět na grafu (*Graf 4.16)*, Yamamotův rekurzivní kód dosahuje jediného relativně dobrého výsledku. Zato Boldi-vigna kód sice s nejlepším výsledkem, ale již ne tak razantním. Zde se jeví faktor k=5 jako nejlepší řešení. Poměrně hezkým výsledkem se chlubí i Yamamotův flag kód, který je o 3% úspěšnější než Wangův, ze kterého vycházel.



Graf 4.16 zobrazující nejefektivnější zástupce kódů pro datový soubor 0-5bitových čísel

Pokud se zaměříme na graf (*Graf 4.17)*, vidíme, že spektrum malých čísel s několika velkými vyhovuje všem kódům mimo Yamamotova rekurzivního. Opět však s nejlepším výsledkem Boldi-Vigna, ale oproti minulému testu teď s nejúspěšnějším faktorem *k=8*.



Graf 4.17 zobrazující nejefektivnější zástupce kódů pro datový soubor normálních čísel

5. Závěr

Ve své práci jsem pochopil a implementoval logiku některých kódů pro bezeztrátovou kompresi dat a připravil je k využití kompresním frameworkem. Prozkoumal jsem většinu logicky vhodných variant pro kompresi určitých kódů pro různá spektra čísel a tím zjistil, kde je která varianta nejúčinnější a také, která varianta dosahuje největší celkové úspěšnosti. Díky tomuto se v dalším používání těchto kódu dá určit, která varianta bude nejvíce vyhovovat datům, na které je budeme aplikovat. Také díky znalosti úspěšnosti nejlepších variant můžeme poznat, že pokud bude potřeba komprimovat data, je nejlepší možností použít Boldi-Vigna kód s faktorem kódování $k = 8$, jelikož dosahuje nejlepšího výsledku s úsporou přes 30% na celém spektru čísel s náskokem 5%, a zároveň nejlepších výsledků dosahuje v kategoriích 0-8 , 8-16 a 16-24 bitů. V posledním bloku 24-32 bitových čísel dosahuje taktéž dobrého výsledku.

Jelikož v posledním bloku čísel se nachází čísla, které po zakódování mají většinou více jak 32 bitů, nebo blízko tomuto počtu, je téměř jisté, že je lepší tyto čísla ponechat v původním znění. Pokud však by se jednalo čistě o čísla, která by nepřesáhla minimální číslo, které by bylo velmi vysoké, čísla je možno upravit odečtením tohoto čísla před kompresí, a poté komprimovat podle vhodné metody, čímž by se dosáhlo velmi efektivní komprese.

Pokud data obsahují pouze opravdu malá čísla, všechny tyto kódy dosahují velmi podobných výsledků. Pokud bychom se rozhodovali podle nejlepší komprese, výběr by měl skončit u Boldi-Vigna kódu s faktorem kódování $k = 5$, či Yamamotovým flag kódem s délkou oddělovače $f = 3$. Ovšem dalším faktorem při rozhodování může být i časová náročnost, či náročnosti výpočtů.

Literatura

[1] Variable-length codes for Data Compression, David Salomon, Springer, 2007

Obsah CD

Adresář	Popis
Bc prace	Implementované kódy v solution Visual Studia 2008
Text	Text bakalářské práce
Data	Soubory testovaných dat